Karlsruhe Institute of Technology

Handling Heterogeneity in Software Architecture Reverse Engineering: A View-Based Approach

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Yves Richard Kirschner

aus Karlsruhe

Tag der mündlichen Prüfung: 30. Januar 2025
Erste Referentin: Prof. Dr.-Ing. Anne Koziolek
Zweiter Referent: Prof. Dr. Uwe Zdun (Universität Wien)

Abstract

In this thesis, we formalize and evaluate how to reverse engineer software architecture models in the context of modern, complex software systems that exhibit heterogeneity in domains, technologies, and interconnectedness.

Understanding and monitoring architectural structures becomes increasingly challenging as software systems evolve to encompass many programming languages, frameworks, and configurations. This research addresses the need for automated techniques to extract architectural models from diverse software artifacts, including source code files, configuration files, and deployment descriptors. The goal is to reconstruct architectural models using a view-based approach that enables an understanding of system structures, components, and interrelationships. This will support modern software systems' performance and security, especially for those based on web service and microservice architectures.

The problem addressed in this thesis is the inadequacy of existing approaches for reverse engineering software architecture models from heterogeneous artifacts across multiple technologies. Key features of modern component-based software systems include independent deployment, loose coupling, and diverse technologies and platforms. These factors introduce complexity and heterogeneity, making the extraction of coherent architectural models a challenging endeavor. Existing approaches sometimes rely on manual documentation or tools that cannot keep pace with accelerating development cycles and increasing technology diversity, resulting in outdated or incomplete architectural representations. The lack of accurate architectural models impedes the ability to analyze and predict system properties, monitor system evolution, and support quality attributes such as performance and security. As a result, an urgent need exists for an approach that handles the inherent heterogeneity and complexity of modern software systems to facilitate understanding and maintenance.

The current suite of reverse engineering tools and approaches is sometimes inadequate to address modern software systems' inherent heterogeneity and complexity. While some tools can support a range of programming languages, they are usually insufficient when confronted with multilingual systems or when tasked with integrating diverse artifacts beyond the source code. These approaches struggle with the different formats, programming languages, and semantics inherent in software artifacts in component-based architectures. Such tools are sometimes inadequate for extracting the necessary information to construct architectural models, especially in systems where services are developed and deployed independently using different technologies and platforms. As a result, there is a lack of practical reverse engineering of software architectures derived from heterogeneous artifacts across technologies. As a result, the architectural models

produced are incomplete and inaccurate, which hinders quality prediction and system analysis.

This thesis introduces a novel, view-based approach for reverse engineering software architecture models from heterogeneous artifacts across technologies called Retriever. This approach uses model-to-model transformations to extract system views from software artifacts, including source code, configuration files, and deployment descriptors. By defining technology-specific extraction rules and applying a knowledge representation model, our Retriever approach can accurately map artifacts to architectural elements. It then applies model composition and refinement techniques to integrate each view into a unified architectural model, supporting coherence and completeness across views. This unified model addresses multiple concerns and provides a foundation for model-based analysis and quality prediction. This thesis contributes to software engineering by developing a new approach that addresses modern software systems' inherent complexity and heterogeneity. Our approach enables the automated generation of architectural models suitable for quality prediction and system optimization, representing an advancement in software engineering.

Our Retriever approach has been empirically evaluated using varying-size open-source projects to assess its applicability and accuracy. The evaluation process involved integrating Retriever into continuous integration pipelines and automating the generation of architectural models. The models were validated for syntactic correctness and analyzed for their ability to abstract complex and heterogeneous code bases into manageable architectural representations. The results indicated that execution times exhibited practical scalability with code base size, indicating suitability for large systems. Appropriate metrics were used to evaluate the extracted structural and behavioral properties' precision, recall, and F_1 scores. The high score achieved indicates that our Retriever approach accurately identified the components and their interrelationships. The results confirm that the approach addresses the challenges inherent in heterogeneous systems and produces valid architectural models. These models are expected to improve understanding, analysis, and quality prediction. As a result, it was shown to have practical applicability in modern software development environments.

Zusammenfassung

In dieser Arbeit formalisieren und evaluieren wir, wie Softwarearchitekturmodelle im Kontext moderner, komplexer Softwaresysteme mit heterogenen Domänen, Technologien und Vernetzungen rückentwickelt werden können.

Das Verständnis und die Kontrolle von Architekturstrukturen werden immer schwieriger, da sich Softwaresysteme weiterentwickeln und viele Programmiersprachen, Frameworks und Konfigurationen umfassen. Diese Forschung befasst sich mit dem Bedarf an automatisierten Techniken zur Extraktion von Architekturmodellen aus verschiedenen Softwareartefakten, einschließlich Quellcodedateien, Konfigurationsdateien und Deployment-Deskriptoren. Ziel ist die Rekonstruktion von Architekturmodellen mithilfe eines Sichtenbasierten Ansatzes, der ein Verständnis von Systemstrukturen, Komponenten und Beziehungen ermöglicht. Dies wird die Performance und Sicherheit moderner Softwaresysteme unterstützen, insbesondere solcher, die auf Web-Service- und Microservice-Architekturen basieren.

Das Problem, das in dieser Arbeit behandelt wird, ist die Unzulänglichkeit existierender Ansätze für das Reverse-Engineering von Softwarearchitekturmodellen aus heterogenen Artefakten über mehrere Technologien hinweg. Zu den Hauptmerkmalen moderner komponentenbasierter Softwaresysteme gehören unabhängige Bereitstellung, lose Kopplung und unterschiedliche Technologien und Plattformen. Diese Faktoren führen zur Komplexität und Heterogenität, was die Extraktion konsistenter Architekturmodelle erschwert. Bestehende Ansätze basieren manchmal auf manueller Dokumentation oder Werkzeugen, die mit den immer schnelleren Entwicklungszyklen und der zunehmenden technologischen Vielfalt nicht Schritt halten können, was zu veralteten oder unvollständigen Architekturdarstellungen führt. Das Fehlen präziser Architekturmodelle erschwert die Analyse und Vorhersage von Systemeigenschaften, die Überwachung der Systementwicklung und die Unterstützung von Qualitätsmerkmalen wie Leistung und Sicherheit. Es besteht daher ein dringender Bedarf an einem Ansatz, der die inhärente Heterogenität und Komplexität moderner Softwaresysteme bewältigt, um das Verständnis und die Wartung zu erleichtern.

Gegenwärtige Reverse-Engineering-Werkzeuge und -Ansätze sind nicht immer in der Lage, mit der Heterogenität und Komplexität moderner Softwaresysteme umzugehen. Einige Werkzeuge unterstützen zwar eine Reihe von Programmiersprachen, sind aber in der Regel unzureichend, wenn es um mehrsprachige Systeme oder die Integration verschiedener Artefakte außerhalb des Quellcodes geht. Diese Ansätze haben mit den unterschiedlichen Formaten, Programmiersprachen und Semantiken zu kämpfen, die den Softwareartefakten in komponentenbasierten Architekturen eigen sind. Solche Werkzeuge sind manchmal

nicht in der Lage, die für die Erstellung von Architekturmodellen erforderlichen Informationen zu extrahieren, insbesondere in Systemen, in denen Dienste unabhängig voneinander unter Verwendung unterschiedlicher Technologien und Plattformen entwickelt und bereitgestellt werden. Dies führt zu einem Mangel an praktischem Reverse-Engineering von Softwarearchitekturen, die aus heterogenen Artefakten über Technologien hinweg abgeleitet werden. Infolgedessen sind die erstellten Architekturmodelle unvollständig und ungenau, was die Qualitätsvorhersage und Systemanalyse erschwerten.

In dieser Arbeit wird ein neuartiger, auf Sichten basierender Ansatz für das Reverse-Engineering von Softwarearchitekturmodellen aus heterogenen Artefakten über Technologien hinweg vorgestellt, der als Retriever bezeichnet wird. Dieser Ansatz verwendet Modell-zu-Modell-Transformationen, um Systemsichten aus Softwareartefakten, einschließlich Quellcode, Konfigurationsdateien und Deployment-Deskriptoren, zu extrahieren. Durch die Definition von technologiespezifischen Extraktionsregeln und die Anwendung eines Wissensrepräsentationsmodells ist unser Retriever-Ansatz in der Lage, Artefakte präzise auf Architekturelemente abzubilden. Anschließend werden Techniken zur Modellkomposition und -verfeinerung angewendet, um jede Sicht in ein einheitliches Architekturmodell zu integrieren, das Konsistenz und Vollständigkeit über alle Sichten hinweg unterstützt. Dieses einheitliche Modell berücksichtigt mehrere Sichten und bildet die Grundlage für modellbasierte Analysen und Qualitätsvorhersagen. Diese Dissertation leistet einen Beitrag zur Softwareentwicklung durch die Entwicklung eines neuen Ansatzes, der die inhärente Komplexität und Heterogenität moderner Softwaresysteme berücksichtigt. Unser Ansatz ermöglicht die automatisierte Erstellung von Architekturmodellen, die für Qualitätsvorhersagen und Systemoptimierungen geeignet sind, und stellt einen Fortschritt in der Softwareentwicklung dar.

Unser Retriever-Ansatz wurde anhand von Open-Source-Projekten unterschiedlicher Größe empirisch evaluiert, um seine Anwendbarkeit und Genauigkeit zu bewerten. Der Evaluierungsprozess beinhaltete die Integration unseres Ansatzes in Softwareentwicklungsprozesse, um die Generierung von Architekturmodellen zu automatisieren. Die Modelle wurden auf syntaktische Korrektheit validiert und hinsichtlich ihrer Fähigkeit analysiert, komplexe und heterogene Codebasen in handhabbare Architekturdarstellungen zu abstrahieren. Die Ergebnisse zeigten eine praktische Skalierbarkeit der Ausführungszeiten mit der Größe der Codebasis, was auf die Eignung für große Systeme hindeutet. Geeignete Metriken wurden verwendet, um die Präzision, die Ausbeute und die F_1-Werte der extrahierten Struktur- und Verhaltenseigenschaften zu bewerten. Die erzielten hohen Punktzahlen zeigen, dass unser Retriever-Ansatz die Komponenten und ihre Wechselwirkungen genau identifiziert hat. Die Ergebnisse bestätigen, dass der Ansatz die mit heterogenen Systemen verbundenen Herausforderungen bewältigt und gültige Architekturmodelle erzeugt. Die erwarteten Vorteile dieser Modelle sind, dass sie das Verständnis, die Analyse und die Vorhersage der Qualität erleichtern. Es konnte gezeigt werden, dass der Ansatz in modernen Softwareentwicklungsumgebungen praktisch anwendbar ist.

Danksagungen

Hinter jeder Seite dieser Arbeit stehen die wertvollen Beiträge vieler wunderbarer Menschen, ohne die diese Dissertation nicht möglich gewesen wäre.

Mein besonderer Dank gilt zunächst meiner Doktormutter, Prof. Dr. Anne Koziolek, deren fachliche Expertise, geduldige Betreuung und stete Motivation diese Arbeit erst möglich gemacht haben. Ihre unzähligen Diskussionen, das wertvolle Feedback und ihr Vertrauen in meine Ideen haben mich sowohl wissenschaftlich als auch persönlich wachsen lassen. Ebenso herzlich danke ich Prof. Dr. Uwe Zdun für die Übernahme des Korreferats und seine konstruktiven Anregungen. Seine Perspektive hat mir geholfen, kritische Punkte zu schärfen und die Arbeit in einen größeren Kontext einzuordnen.

Ich möchte mich bei Axel und Kiana bedanken, die mich schon während meines Studiums für die Wissenschaft begeistert und den Grundstein für diesen Weg gelegt haben. Ohne eure Unterstützung während meines Studiums wäre ich heute nicht da, wo ich bin. Für die engagierte Unterstützung bei der Umsetzung und Erprobung möchte ich mich ganz besonders bei meinen studentischen Hilfskräften Florian, Moritz und Nico bedanken. Ihr habt gezeigt, was Begeisterung und Neugier alles bewegen können und mir so manche Hürden genommen.

Mein besonderer Dank gilt meinen Kollegen aus der Programmierlehre – Dominik, Haoyu, Nils, Maximilian und Timur. Ihr Teamgeist, die kreativen Unterrichtsideen und der gemeinsame Spaß an der Wissensvermittlung haben mir immer wieder neue Energie gegeben. Für die Unterstützung bei der Implementierung dieser Arbeit möchte ich mich bei Larissa, Nikolas und Stephan bedanken, die mit ihrem Fachwissen geholfen haben, die Arbeit in ihre endgültige technische Form zu bringen. Ebenso möchte ich mich bei Jan, Sebastian und Sophie für die spannenden Reisen zu Konferenzen und Forschungsaufenthalten bedanken – eure Gesellschaft und die inspirierenden Gespräche haben jede Reise zu einer bereichernden Erfahrung gemacht. Mein besonderer Dank gilt ebenso Frederik, Sandro und Tobias. Ihr habt mit mir dunkle Verliese durchquert und mythische Kartenspiele gespielt, wodurch selbst endlose Arbeitstage zu einem echten Abenteuer wurden. Für die reibungslose Organisation im Hintergrund danke ich Claudia vom Sekretariat. Deine Zuverlässigkeit und Freundlichkeit haben mir in stressigen Phasen den Rücken freigehalten. Den weiteren Mitgliedern der Forschungsgruppe – Erik, Jörg, Lars, Sebastian, Snigdha und Thomas – danke ich für die inspirierende Zusammenarbeit, die produktiven Diskussionen und die stets offene Tür, die unsere Arbeitsatmosphäre so besonders gemacht haben.

Meinen Freunden Felix, Tobias und Tom danke ich für die moralische Unterstützung, die langen Gespräche über alles Mögliche (und Unmögliche) und dafür, dass ihr mir immer den Kopf frei gehalten habt. Von klein auf haben mich Gertrud, Conny und Franz begleitet und unterstützt – dafür bin ich ihnen unendlich dankbar. Vor allem ihre Hilfe während der Schulzeit hat mir das Studium überhaupt erst ermöglicht.

Meine Eltern Kerstin und Friedbert sowie meine Schwester Inès haben mir von klein auf den Rücken gestärkt, mich ermutigt und mir bedingungslos vertraut. Dafür, dass ihr immer an mich geglaubt habt – auch wenn ich selbst gezweifelt habe – danke ich euch von ganzem Herzen.

Mein größter Dank gilt meiner Frau Fabienne und unserer Tochter Amaya. Fabienne, du hast jeden Schritt dieser Reise mit Geduld, Liebe und Humor begleitet. Ohne deine Zuversicht in stressigen Schreibphasen und dein Verständnis für lange Arbeitstage wäre dies nicht möglich gewesen. Amaya, dein Lachen hat mich jeden Tag daran erinnert, was wirklich zählt. Ihr seid mein Anker und meine größte Motivation.

Contents Overview

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

List of Definitions

List of Acronyms

ADL Architecture Description Language. 31–33, 74, 115, 118, 180, 235

API Application Programming Interface. 21, 22, 27, 59, 110, 111, 128, 153–156, 164, 166–169, 173, 223

AST Abstract Syntax Tree. 43, 112, 229

ATL Atlas Transformation Language. 27, 28, 89, 115

BNF Backus-Naur Form. 71–73, 77

CBSE Component-Based Software Engineering. 19, 20, 35

CBSS Component-Based Software Systems. 234

CD Continuous Delivery. 107

CI Continuous Integration. 107

CI/CD Continuous Integration and Continuous Delivery. 7, 12, 23, 43, 107–109, 113, 123, 124, 140, 143–145, 151, 161, 184, 188, 238

CLI Command-Line Interface. 109, 112, 114, 161, 174, 182, 184

CLRS Communication Link Resource Specification. 116

CMOF Complete Meta Object Facility. 25

CORBA Common Object Request Broker Architecture. 19

CSV Comma-Separated Values. 50, 110

CVE Common Vulnerabilities and Exposures. 23, 24, 39, 85–87, 114, 161, 175, 182, 184, 185

CVSS Common Vulnerability Scoring System. 24, 39, 194

CWE Common Weakness Enumeration. 23, 24, 39

DOI Digital Object Identifier. 241

DSL Domain-Specific Language. 25–27, 33, 211

EJB Enterprise Java Bean. 19

EMF Eclipse Modeling Framework. 12, 26–28, 108, 109, 111, 139–141, 143–145, 180, 211

EMOF Essential Meta Object Facility. 26

EPL-2.0 Eclipse Public License 2.0. 108, 115

GPL General-Purpose Language. 89

GQM Goal Question Metric. 12, 123, 125, 126, 140, 177, 178, 192, 193

GUI Graphical User Interface. 109, 229

HTML Hypertext Markup Language. 155, 216

HTTP Hypertext Transfer Protocol. 20–22, 57–59, 154

IaaS Infrastructure as a Service. 21

IDE Integrated Development Environment. 43, 109, 110, 112

ISO International Organization for Standardization. 44, 53

JAX-RS Jakarta RESTful Web Services. 12, 13, 113, 128, 129, 234

JDT Java Development Tools. 110, 113

JSON JavaScript Object Notation. 5, 50, 111, 113, 155

JSP Jakarta Server Pages. 229

JVM Java Virtual Machine. 111

KDM Knowledge Discovery Metamodel. 41

LoC Lines of Code. 12, 126, 127, 130, 142, 143, 145–151, 160, 171, 178–181, 185, 186

LPG LALR Parser Generator. 141

M2M Model-to-Model. 28

MBE Model-Based Engineering. 209

MDE Model-Driven Engineering. 25, 228

MDQP Model-Driven Quality Prediction. 34

MDRE Model-Driven Reverse Engineering. 34, 39–42, 225, 228–230

MDSD Model-Driven Software Development. 24, 25, 28

MOF Meta Object Facility. 25, 26, 222

MPS Meta Programming System. 215

MTL Model Transformation Language. 28

NIST National Institute of Standards and Technology. 23

NVD U.S. National Vulnerability Database. 23, 39, 84, 86, 87, 174, 182, 184

OCL Object Constraint Language. 12, 141, 144, 145, 178, 180, 224

OMG Object Management Group. 25, 26, 41

OSGi Open Services Gateway initiative. 19

PaaS Platform as a Service. 21

PCM Palladio Component Model. 11, 12, 33–38, 41, 42, 108–112, 115–119, 140–144, 161, 174, 180, 194, 217, 232, 235

PMX Performance Model eXtractor. 88

QVT Query/View/Transformation. 27, 28, 89, 115

RAM Random-Access Memory. 142

REST Representational State Transfer. 21, 22, 57–59, 74, 84–86, 113, 128, 129, 153–156, 163–165

Retriever **R**everse **E**ngineering **T**echnique with **R**efinement **I**ntegration of **E**xtracted **V**iews with **E**lements and **R**elationships. 5, 11, 49, 50

SaaS Software as a Service. 21

SAR Software Architecture Reconstruction. 204

SARE Software Architecture Reverse Engineering. 216

SAST Static Application Security Testing. 43

SEFF Service Effect Specification. 36, 37, 42, 43, 98, 112, 116, 127, 159, 170

SOA Service-Oriented Architecture. 21, 227, 228

SOAP Simple Object Access Protocol. 20, 21

SoMoX SOftware MOdel eXtractor. 42, 79, 80, 82, 112

SPDX Software Package Data Exchange. 110

SQL Structured Query Language. 43, 111, 113, 211

SUM Single Underlying Model. 31, 207, 209, 210

SysML Systems Modeling Language. 215, 231

T2M Text-to-Model. 28

UML Unified Modeling Language. 12, 20, 25–28, 32, 33, 41, 44, 53, 61–63, 95, 107–109, 111, 118, 140, 141, 144, 157, 158, 180, 203, 207, 212, 213, 215, 217, 223–225, 228, 230–232

URI Uniform Resource Identifier. 21, 58, 94, 159

URL Uniform Resource Locator. 56–58, 61, 130

W3C World Wide Web Consortium. 20

WSDL Web Services Description Language. 20, 21

XACML eXtensible Access Control Markup Language. 38, 194

XML Extensible Markup Language. 20, 21, 111, 113, 124, 229

YAML YAML Ain't Markup Language. 5, 50, 112, 113, 124

Part I.

Prologue

1. Introduction

In today's digital age, software systems are rapidly increasing in domain complexity [Ban+93; BDS98], technology heterogeneity [San+14; AC18; Ter19; Noa+22], and interconnectedness [AWY13; MH16; JT18; MOT18], posing challenges to those charged with managing and operating them [Rya99; CC03; MNH15; Mur11]. Understanding and managing these systems is more complex than ever; supporting their functionality [Crn01; Oqu16], scalability [SHS09; YX16; OT19], and security [KBN12; Fur+16; Duc+17] is imperative. Traditional approaches to documenting [Bac+00; Cle14] and analyzing [SWC93; FRS16; Wan+23] software architectures are sometimes insufficient when faced with the wide variety of technologies and artifacts [Pfe20] that characterize today's systems. In this thesis, "artifacts" refers to source code files, configuration files, and deployment descriptors [UML] from which architectural models are derived.

Consider the challenge of assembling a puzzle, where each piece comes from a different set and exhibits variability in shape, size, and pattern. This is analogous to reverse engineering the architecture of a software system built with multiple languages, frameworks, and configurations. The heterogeneity of formats presents a challenge to extracting a coherent architectural model. Figure 1.1 illustrates the puzzle analogy used for our motivation.

However, this challenge provides an opportunity to consider a novel approach. A view-based approach to reverse engineering allows for multiple system views, facilitating the reconstruction of an architectural model. In this thesis, the term "view" describes the different perspectives from which a software system can be examined to gain insight into its architectural structure [I42010]. Each view focuses on a particular aspect or concern, such as structural components, behavioral interactions, or deployment configurations. By extracting and integrating these individual views, a unified model can be constructed to understand the system's architecture.

This reverse engineering approach employs model-to-model transformations to facilitate the translation of diverse software artifacts into a unified architectural representation. Model-to-model transformations involve converting one model into another, thereby facilitating the translation of diverse software artifacts into a unified architectural model. Model composition techniques are then applied to integrate these views seamlessly, eliminating overlap and supporting coherence throughout the model. This approach addresses the complexity introduced by different technologies and improves our ability to analyze and predict system properties, such as performance.

1.1. Motivation

Software systems are everywhere, from the personal smartphone applications [Bal+06] that have become an integral part of our lives to the complex infrastructures that underpin global networks [PH05]. These systems are not static; they constantly evolve, becoming critical to various operations. Understanding and managing these systems poses challenges in scientific and industrial contexts, particularly concerning their underlying architectures [Mur11; Oqu16]. The architecture of a software system is fundamental because it defines the system's structure, components, and interconnections and directly impacts performance, scalability, maintainability, and security [Kaz+94; Bog+19].

The traditional approach to developing and maintaining architectural models has been time-consuming and potentially inaccurate. Software architects and engineers typically rely on documentation, including diagrams and textual descriptions, which can quickly become outdated as the system evolves [LSF03; Agh+19; Agh+20]. Manual approaches cannot keep up with the accelerated development cycles of modern software, resulting in a mismatch between the documented architecture and the implemented system [Ali+17].

In response to these challenges, there has been a shift toward automated extraction of architectural models from software artifacts [MWT95; Mül+00]. Automated architectural model extraction offers advantages in speed and accuracy, enabling the modeling of complex systems that would be impractical to document manually [CS10]. However, automating architectural model extraction presents several challenges, particularly in the context of web services or microservice systems [CD07]. These systems are heterogeneous, encompassing different platforms, programming languages, and underlying frameworks [RAZ17]. The independent deployment and loose coupling of services further complicate the extraction process, as services may be developed and maintained independently using different technologies and platforms [CDC11].

The heterogeneity of these artifacts leads to differences in syntax and semantics [Pfe20], which poses a challenge to developing tools that can extract models from all types of software artifacts [Ali+17]. To illustrate, the semantics embedded in source code may differ from those embedded in configuration files or deployment descriptors. In addition, software artifacts sometimes lack information, requiring the incorporation of additional data to construct an architectural model. The complex relationships between components and configurations require analysis techniques to accurately represent these elements in a model [RAZ17].

Current state-of-the-art approaches are sometimes insufficient to address these challenges [RAZ17]. While some tools can support a range of programming languages, they are insufficient when confronted with multilingual systems or integrating different types of artifacts beyond source code [Ali+17]. It is essential to develop an approach capable of reverse engineering software architectures from diverse artifacts across technologies, integrating different views into a unified model that addresses the various concerns about the system.

Our approach – named Retriever– aims to fill this gap by introducing a view-based approach to reverse engineering software architecture models. This approach is based on two concepts. First, it employs a knowledge representation model for software architecture views capable of encompassing the different technologies, formats, and languages in various artifacts. This is achieved through model-to-model transformations that reconstruct views from system artifacts, considering technology-specific relationships and concepts. Second, it provides a framework for integrating individual views into a unified architectural model. This framework uses model transformation and composition techniques to map concepts between view-specific models and incorporate them into a unified model based on an existing metamodel.

Our Retriever approach addresses modern software systems' inherent heterogeneity and complexity by leveraging these concepts. This approach enables the extraction and integration of multiple views, each representing different concerns and views of the system, into a unified architectural model. This model serves two purposes. First, it facilitates the understanding and documentation of the architectural design. Second, it provides a basis for predicting quality attributes such as performance and security using dedicated tools.

1.1.1. Challenges

The process of reverse engineering software architectures for modern web services and microservices systems presents several challenges, mainly due to these systems' inherent heterogeneity and complexity. One of the difficulties arises from the different formats, programming languages, and semantics of the software artifacts from which architectural models must be extracted. Such artifacts can take many forms, including source code written in multiple programming languages, configuration files, and deployment descriptors. Each of these artifacts has its own characteristics.

The independent deployment and loosely coupled nature of services in microservice architectures further complicate the extraction process. Services are developed and deployed independently, potentially using different technologies and platforms. The result is a heterogeneous ecosystem where components interact through well-defined interfaces but lack a centralized structure. The decentralized nature of these systems presents a challenge to building a coherent architectural model that accurately reflects the system's structure and behavior.

The heterogeneous nature of the syntactic and semantic constructs inherent in different programming languages and configuration formats is a barrier to developing a unified extraction approach. Automated tools must be able to account for these differences to parse and interpret the artifacts. To illustrate, the semantics of source code in a statically typed language, such as Java, differ from those in a dynamically typed language, such as JavaScript. These two forms of expression differ from the semantics of configuration files written in formats such as YAML Ain't Markup Language (YAML) or JavaScript Object Notation (JSON). The need for an extraction mechanism capable of handling multiple languages and formats arises from the diversity of relevant material.

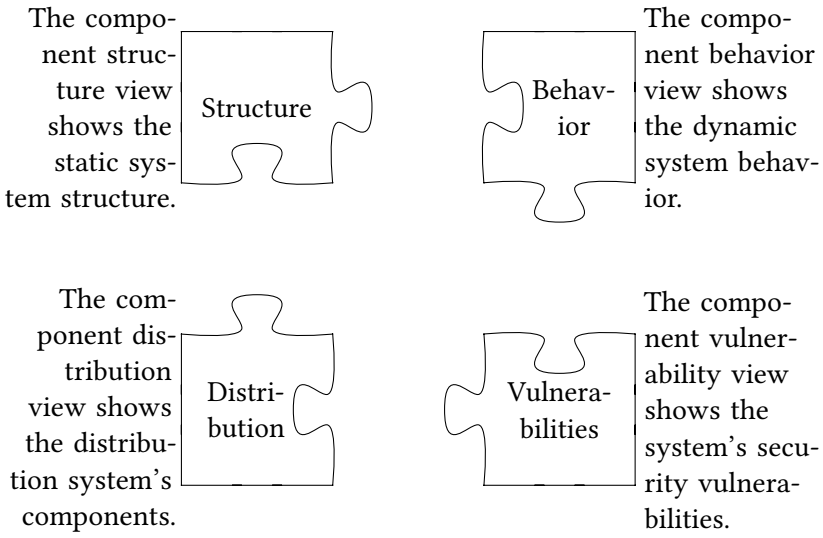


Figure 1.1.: The puzzle analogy is used for motivational purposes. In this analogy, four views of a system are represented as puzzle pieces that illustrate the system's complexity under study. Each piece represents a different system view, including structural elements, behavioral patterns, potential vulnerabilities, and distributional views. The goal is to construct a coherent and unified architectural model by integrating the pieces into a unified whole.

Another challenge is incomplete information in artifacts. It is not uncommon for source code to lack explicit deployment details or to omit specific architectural concerns. In such cases, additional sources of information may be needed to construct a model. The relationships between components, such as dependencies, communication patterns, and configurations, can be complex and implicit, requiring analysis techniques to identify and accurately represent them in the model.

Most existing reverse engineering approaches were designed to work in the context of homogeneous systems or to support a limited set of technologies. As a result, they are typically ill-suited for implementation in today's heterogeneous environments. In addition, research is needed to address the challenge of supporting systems composed of multilingual source code and different artifact types. The relationship between architectural elements and source code is typically not explicitly defined, but instead embedded in the developer's understanding. This introduces ambiguity in reverse engineering, which must be addressed by applying accurate techniques.

Ensuring coherence and completeness across different system views is a challenge. Stakeholders may have other concerns, so the architectural model must integrate these distinct views into a unified whole. Model transformations and compositions must support semantic preservation and accurately represent the system's architectural structure. This requires techniques to manage the inherent complexity of combining multiple models and views.

Scalability is a challenge when dealing with large, real-world systems. The reverse engineering approach must be applicable enough to accommodate the analysis of large code bases and various artifacts while avoiding undue computational overhead. The system

should also be able to abstract away details, thereby reducing complexity while retaining the architectural information necessary to facilitate analysis and quality prediction.

1.1.2. Application Scenarios

Our Retriever approach offers advantages in several practical software engineering scenarios, particularly in complex and diverse systems and the need for accurate architectural understanding. An application scenario is the reverse engineering of component-based systems. It is not uncommon for organizations to maintain large software systems that have been developed over a long period of time. Such systems sometimes lack up-to-date documentation due to the constant changes and personnel changes inherent in the development process, resulting in a lack of documentation. Our approach allows engineers to extract architectural models from multiple sources, including code bases, configuration files, and deployment scripts. Integrating these views provides an understanding of the system's architecture, which benefits maintenance, and refactoring efforts.

In addition, our Retriever approach applies to microservice-based architectures that are inherently distributed and use multiple technologies. Independent deployment, loose coupling of services, and using different programming languages and platforms are challenges to understanding the overall system structure. The approach addresses this problem by extracting architectural models from heterogeneous artifacts, enabling developers to visualize component interactions and dependencies. This understanding is essential for optimizing microservice architectures' performance and security.

In the context of Continuous Integration and Continuous Delivery (CI/CD) pipelines, our Retriever approach can be integrated to automate the generation of architectural models with every code commit. This constant architectural review ensures that the evolving codebase remains up-to-date with the intended architectural design and prevents architectural drift. Automating model generation enables early detection of integration issues, facilitates collaboration between development teams, and improves overall software quality by providing up-to-date architectural documentation.

In addition, our Retriever approach provides benefits in the context of software quality prediction. Creating architectural models that include structural and behavioral properties enables the application of model-based analysis techniques to predict attributes such as performance or attack propagation. Such predictions facilitate informed decision-making regarding architectural adaptations and resource allocation, thereby contributing to developing high-performance and secure software systems.

Another potential application scenario is the analysis of security concerns. In the modern era, software systems are forced to address many security concerns, underscoring the need to understand the architectural implications of security vulnerabilities. Our approach can integrate security-related artifacts and trace links to map vulnerabilities in code to specific architectural elements. This integration allows for an analysis of potential security risks within the architectural model, enabling the development of targeted mitigation strategies and strengthening the overall security posture of the system.

In the context of systems comprising multilingual codebases and multiple technologies – a common occurrence in large enterprise environments – our approach consolidates diverse artifacts into a coherent architectural model. This capability is valuable in mergers and acquisitions, where consolidating diverse systems requires understanding each system's architecture. The approach provides a unified view that can be used to plan integration strategies, reduce redundancies, and support compatibility between diverse systems.

1.2. Research Objective

This thesis aims to formalize, develop, and evaluate our novel Retriever approach for reverse engineering software architecture models from heterogeneous artifacts using multiple system views. This approach addresses the challenges posed by the increasing complexity and heterogeneity of modern software systems, especially those built using web services and microservices architectures. The goal is to develop a practical approach for extracting architectural models from diverse artifacts across different technologies and integrating the various views into a unified model that addresses multiple concerns.

To achieve this goal, the thesis focuses on two areas. The preliminary phase involves the construction of a knowledge representation model capable of processing diverse artifacts, including those in different formats, written in various languages, and imbued with different meanings. This requires the creation of model-to-model transformations that consider technology-specific relationships and concepts. As a result, our approach can accurately extract structural and behavioral views from various software artifacts, including source code, configuration files, and deployment descriptors.

Second, the goal is to develop a framework for integrating individual views into an architectural model. This requires applying model transformation and composition techniques to map concepts between view-specific models and harmonize diverse views into a unified representation based on an existing metamodel. The integration process supports coherence and completeness across different views, enabling stakeholders to understand the software system architecture.

This thesis aims to contribute to the field by addressing these areas. First, it proposes a technique for transforming reverse-engineered models into accurate representations that closely approximate the component-based software system architecture. Second, it presents an approach that combines model-driven reverse engineering processes to capture information across multiple views, thereby resolving overlaps and incoherence between them.

The thesis includes formalizing and developing our view-based reverse engineering approach, including implementing extraction, refinement, and finalization rules that can be applied to different technologies and artifacts. An open-source implementation facilitates the adoption of the approach and further research. Our Retriever approach is empirically evaluated using real-world projects of varying size and complexity to evaluate its applicability and accuracy. The approach's applicability in generating valid architectural

models that abstract from source code and accurately represent structural and behavioral properties is evaluated using precision, recall, and F_1 score metrics.

The goal is to develop a reverse engineering approach that not only automates the generation of architectural models suitable for quality prediction but also improves the understanding and governance of complex software systems. This facilitates the evaluation of system attributes, the prediction of quality characteristics, and the making of well-informed design decisions by software architects and engineers. As a result, performance, scalability, maintainability, and security can be optimized.

1.2.1. Research Goals and Questions

Today's software systems are increasingly complex, heterogeneous, and evolving, posing challenges to understanding and management. A software system's architectural design, which describes its structural framework, components, and interrelationships, impacts its performance, scalability, maintainability, and security. Standard approaches to manually creating and maintaining architectural descriptions are time-consuming, error-prone, and sometimes inaccurate due to the inherent limitations of the human mind. Therefore, there is an urgent need to develop automated techniques for extracting architectural models from existing software artifacts to facilitate the understanding and managing these complex systems.

This research aims to develop a practical approach for reverse engineering software architecture models from heterogeneous artifacts that span multiple concerns and technologies. This addresses the challenges posed by the diversity of formats, languages, and semantics inherent in modern software systems, particularly those using web services or microservices architectures. These systems typically consist of independently deployed and loosely coupled services on different technologies and platforms. This poses a challenge to extracting a coherent and unified architectural model. The definition of the research goal enables the formulation of research questions (RQ). The two research questions to be answered by implementing the approach are:

RQ1 What is an applicable and accurate approach to reverse engineering software architecture for diverse artifacts across technologies?

RQ2 How can different views of a software system be integrated into a unified model that addresses diverse concerns?

The first research question (RQ1) seeks to identify a practical approach for reverse engineering software architectures from diverse artifacts across different technologies. This requires the development of a knowledge representation model that can accommodate the heterogeneity of software artifacts, including source code in various programming languages, configuration files in different formats, and automation scripts. The approach should be able to transform these diverse artifacts into a common representation suitable for architectural modeling, considering technology-specific relationships and concepts.

The second research question (RQ2) concerns integrating different views of a software system into a unified model that addresses other concerns. A software system can be viewed from various views, each focusing on a specific concern, such as structure, behavior, usage, or deployment. Integrating these different views into a unified architectural model requires the development of a framework that can map concepts between the view-specific models and compose them into a unified representation. The framework should employ model transformation and composition techniques to reconcile the different views, resolve incoherence, and support coherence across the unified model.

Two research questions have been formulated to achieve the research goal. This thesis aims to advance the state of the art in software architecture reverse engineering by addressing the above research questions. It presents an approach that reconstructs component-based software architectures and combines model-driven reverse engineering processes to capture information across multiple views. This approach facilitates the automated generation of architectural models suitable for quality prediction and can be used with specialized tools, thereby extending the scope of architectural documentation beyond mere documentation.

1.2.2. Context and Assumptions

The complex nature of today's software systems, especially those using web services and microservices architectures, poses a challenge to understanding and monitoring their architectures. These systems are characterized by heterogeneity in programming languages, technologies, and deployment environments. This heterogeneity can be attributed to three factors: the independent development and deployment of services, diverse technologies, and the inherent loose coupling in microservice architectures.

A challenge in reverse engineering architectural models from such systems is the various formats and semantics of the software artifacts involved. These artifacts, including source code files, configuration files, build scripts, and deployment descriptors, exist in different formats and languages, each with its own characteristics. This diversity challenges the development of automated tools capable of extracting architectural models coherently across technologies.

The Retriever approach addresses these challenges by using multiple views extracted from heterogeneous artifacts to reconstruct a unified architectural model. The assumption is that, despite the diversity of artifacts, it is possible to define a common knowledge representation that can capture the architectural elements across different technologies. This requires formulating technology-specific extraction rules that map artifacts to architectural concepts, considering each technology's specific characteristics and relationships.

Another assumption is that the integration of individual views into a unified model can be achieved by applying model transformation and composition techniques. By defining rules for refinement and composition, our approach resolves incoherence and overlaps between views, ensuring that the resulting model accurately reflects the system architecture. This process assumes that views can be composed based on common concepts and that the

resulting model can be evaluated against an existing metamodel, such as the Palladio Component Model (PCM).

The Retriever approach assumes that automated model extraction and integration can be scaled to accommodate real-world systems of varying size and complexity. This indicates that the applicability of the extraction, refinement, and composition processes remains up-to-date even as the number of artifacts and the complexity of their relationships increase. Our approach to various projects provides empirical support for this assumption and demonstrates its ability to generate valid architectural models.

The Retriever approach assumes that the extracted models are detailed to support subsequent analysis, such as quality prediction or performance evaluation. This requires that the models capture not only the structural properties of the architecture but also the behavioral properties and resource interactions. The approach must include the ability to represent component interactions across different deployment nodes and service impact specifications, as this is an essential aspect of the approach.

1.3. Contributions

This thesis introduces Retriever, a novel approach for reverse engineering software architecture models from heterogeneous artifacts across multiple technologies. This approach addresses the inherent complexity of modern software systems – web services and microservices systems – characterized by independent deployment, loose coupling, and heterogeneous technologies and platforms. Based on this formalization, development, and evaluation, we make the following contributions (C) that go beyond the state of the art:

C1 A technique for transforming reverse-engineered models into accurate representations that approximate the component-based software system architecture.

C2 A technique that combines model-driven reverse engineering processes to capture information while processing multiple views and resolving overlaps between them.

Our first contribution (C1) is developing a knowledge representation model that can accommodate different artifacts, including source code files and configuration files, and build scripts in various formats and languages. This model employs model-to-model transformations that consider technology-specific relationships and concepts, thereby enabling the extraction of structural and behavioral views from these heterogeneous artifacts. Our Retriever approach can accurately map software artifacts to architectural elements by defining extraction rules tailored to specific technologies and capturing components, interfaces, dependencies, and deployment configurations.

Our second contribution (C2) is the framework for integrating individual views into a unified architectural model. This is achieved by implementing model composition and refinement techniques that harmonize different views of a software system. Refinement rules support the coherency and completeness of each view, while composition rules

facilitate the integration of these refined views into a model based on an existing metamodel. The unified model addresses multiple concerns and views, representing the software architecture that supports model-based analysis and quality prediction.

Our Retriever approach formalizes model-driven reverse engineering processes, facilitating the integration and unification of architectural descriptions from multiple views. It extends the scope of architectural documentation by automating the creation of architectural models suitable for quality prediction. The metamodel-independent approach uses model transformations to link model-providing processes with model-consuming processes, improving adaptability to diverse software systems.

Technically, our Retriever approach is implemented using the Eclipse Modeling Framework (EMF). It is compatible with several widely used technologies, including Docker, ECMAScript, Gradle, Jakarta RESTful Web Services (JAX-RS), Maven, and Spring. The extraction rules for these technologies are designed to identify and map architectural elements from different software artifacts, considering each technology stack's characteristics. The open-source approach is available as an Eclipse plug-in, facilitating integration into existing development workflows and continuous integration pipelines.

1.3.1. Evaluation

The objective of evaluating our Retriever approach was to validate its applicability and accuracy in reverse engineering software architectures derived from heterogeneous artifacts. The Goal Question Metric (GQM) approach was used to structure the evaluation, where specific goals, questions, and metrics were defined to validate our approach.

To determine the applicability of our Retriever approach, a set of extraction rules was implemented for several widely used technologies, including Maven, Gradle, Docker, Spring Boot, ECMAScript, and JAX-RS. These technologies are examples of the frameworks and tools prevalent in today's component-based systems, particularly those related to web services and microservices. The most relevant open-source projects were selected based on their popularity on GitHub, considering commits, contributors, forks, and stars. The systems had a range of sizes and complexity, providing a set of case studies for evaluation.

The Retriever approach was integrated into a CI/CD pipeline using GitHub Actions. The automatically generated architectural models were evaluated for syntactic correctness using PlantUML for Unified Modeling Language (UML) diagrams, And the Object Constraint Language (OCL) constraints were evaluated for PCM models. The models were free of errors or warnings, demonstrating that our approach can produce syntactically correct architectural models.

The Retriever approach's scalability was evaluated by analyzing execution times on several projects. The results showed an increase in execution time with the size of the input code base, measured in Lines of Code (LoC). To illustrate, the largest project, with over 38 000 LoC, had an execution time of approximately 16 seconds, which is considered

acceptable. This practical scalability indicates that our approach suits larger, real-world systems well. Abstraction capacity was evaluated by comparing the number of source files to the number of components in the generated architectural models. The results showed a reduction in complexity, with code bases abstracted into a manageable number of components.

The accuracy of our Retriever approach was evaluated by comparing the automatically generated models with manually generated gold standard models for a subset of the projects. Quantitative metrics assessed precision, recall, and the F_1 score of structural and behavioral properties. The results indicated that our approach could accurately identify most components and their interrelationships, achieving F_1 scores of up to 0.94 for structural properties and up to 0.90 for behavioral properties. However, our approach showed shortcomings in identifying components not directly related to the system's functions, such as configuration-related ones. It also identified instances of incorrect mappings between interfaces and components.

The accuracy of the results was found to depend on the initial extraction phase, in particular, the technology-specific extraction rules. Frameworks with stringent implementation requirements, such as Spring, provided better support for accurate extraction than frameworks such as JAX-RS. Therefore, it was suggested that creating project-specific guidelines could improve the accuracy of reverse-engineered models. The effort required to define project-specific rules was also examined in this context. The results indicate that the effort needed to extend the rules to include project-specific characteristics has a practical advantage.

1.3.2. Expected Benefits

Our Retriever approach offers advantages for reverse engineering software architecture models, especially for complex and heterogeneous systems such as web services and microservices. One advantage is its ability to accommodate heterogeneous artifacts across technologies, formats, and programming languages. By leveraging a knowledge representation model that spans multiple artifacts, our approach automates the extraction of architectural models from various sources, including source code, configuration files, and deployment descriptors. This automation reduces reliance on manual processes that are sometimes time-consuming and error-prone, thereby increasing the accuracy of the architectural models produced.

Another benefit is integrating multiple views into a unified architectural model. Our approach uses model transformation and composition techniques to facilitate the mapping of concepts between different view-specific models. This integration addresses different concerns and technologies within the software system, providing an understanding of its architecture. Combining structural and behavioral views allows for detailed representation, which is essential for in-depth analysis and interpretation of complex systems.

It is also worth noting that our Retriever approach is practically scalable. The execution time for various open-source projects has been shown to scale with the size of the input

code base, as evidenced by evaluations of these projects. Our approach was evaluated regarding total execution time, from initial initiation to final shutdown. This indicates that our approach is well suited to handling large systems, such as those in industrial applications. As a result, performance remains within acceptable limits, even as system complexity increases. Our approach abstracts from the source code to produce valid architectural models.

Retriever increases software development and maintenance productivity and efficiency by automating the reverse engineering process. It reduces the need for manual effort, decreases the likelihood of human error, and supports the idea that architectural models remain up-to-date with the evolving system. The implementation's open-source nature allows for adaptability, enabling practitioners to customize the extraction, refinement, and finalization rules to suit specific technologies and project needs.

In addition, our approach enables the integration of architectural modeling into continuous integration and deployment pipelines. Automating model generation with each code commit enables continuous architectural review and documentation. This integration helps prevent the gradual erosion and drift of architectural principles over time, supporting continuous alignment between the system architecture and its implementation. This supports the idea that architectural considerations remain an aspect of the development process, facilitating the creation of maintainable systems.

In addition, the Retriever approach's ability to generate models suitable for quality prediction extends its applicability beyond mere documentation. By accurately capturing structural and behavioral properties, our approach provides a foundation for quality analysis, including performance prediction and security evaluation. This modeling approach provides the information needed to make informed decisions about system optimization and improvement, finally contributing to developing higher-quality software systems.

1.4. Thesis Outline

Part I, the Prologue, consists of Chapters 1 and 2 and presents the basic concepts and motivations for the thesis. Chapter 1, the Introduction, addresses the motivations for the thesis, articulates the challenges and application scenarios, sets the research objectives, and outlines the contributions and structure of the thesis. It is further divided into sections that provide an overview of the challenges, application scenarios, research objectives, questions, context, assumptions, evaluation methods, and expected benefits of our Retriever approach. Chapter 2, Foundations and Terminology, lays the terminological groundwork and discusses theories and methodologies spanning software architecture, component-based engineering, and model-driven development, including explanations of architecture description languages and reverse engineering processes.

Part II focuses on our presented view-based reverse engineering approach, encapsulated in Chapters 3, 4, 5 and 6. Chapter 3 introduces the reverse engineering approach, illustrates the hourglass paradigm process, and techniques for extracting components and completing

the architecture. Chapter 4 elaborates on extracting various structural and behavioral views from software artifacts, introducing a formal rule concept and its application to view generation and traceability. Chapter 5 elaborates on the composition and refinement of these views into coherent models, discussing the techniques and frameworks involved in this process. Chapter 6 integrates a quality prediction model into the reverse engineering pipeline and discusses the approaches for model-based view extraction, composition, refinement, and final validation of the views.

Part III focuses on the Retriever approach's experimental evaluation and theoretical implications, as presented in Chapters 7 and 8. Part III reports on the empirical evaluation of the proposed approaches through evaluation scenarios, experimental design, and results, focusing on their applicability and accuracy. Chapter 7 discusses the results regarding their validity and implications for software engineering, including an analysis of potential threats to our results' validity.

Part IV, the Epilogue, in Chapters 9 and 10, synthesizes the thesis concerning the existing literature and outlines future directions. Chapter 9 reviews related work, provides a taxonomy of reverse engineering approaches, and discusses the relationship between the current research and existing models and tools. Chapter 10 concludes the thesis by summarizing the research scope, limitations, and future work, describing the assumptions, constraints, and limitations, and providing a brief overview of further approaches and potential future work.

2. Foundations and Terminology

This section establishes the foundation for discussing software architecture and vulnerabilities, component-based and model-driven engineering, architecture description languages, reverse engineering, and evaluation foundations and terminology. By exploring these foundational concepts, we aim to situate our research within the scientific discourse. Furthermore, we illustrate how our view-based reverse engineering approach is consistent with these disciplinary domains.

> The subsequent sections are based on the following authored publications: [SK19; Kir21; Kir+23a; Kir+23b; Gst+24; Kir+24a; Kir+24b]

2.1. Software Architecture

Software architecture is a discipline within software engineering that defines a software system's high-level structure and organization. According to Definition 1, it involves making design decisions about the system's components, their interrelationships, and how these elements are mapped to execution environments [Som18]. These decisions are for developing scalable and secure software systems and serve as a blueprint for subsequent development and maintenance. The architectural design must address the system's inherent complexity while supporting it and meeting functional and non-functional requirements. The functional requirements of a system relate to its intended behavior, while the non-functional requirements relate to performance, security, and reliability [PW92; BCK22].

> **Definition 1** (Software System)
>
> "A software system is a group of devices or software that form a network to serve a common purpose. The system usually consists of separate software and configuration files, or may include system documentation that describes its structure." (Sommerville [Som18])

According to the standard in Definition 2, software architecture can be defined as the fundamental organization of a system, embodied in its components and their relationships to each other and the environment. It also includes the principles that guide its design and

evolution. This definition emphasizes the importance of both structural and guiding principles in the context of software architecture. It indicates that architectural considerations extend beyond the static structure of a system to include its relationships and interactions [Ric20; I42010].

Definition 2 (Software Architecture)

"Fundamental concepts or properties of an entity in its environment and governing principles for the realization and evolution of this entity and its related life cycle processes." (International Organization for Standardization [I42010])

Most architectural decisions are typically made in the early stages of the software development process, coinciding with the requirement analysis phase. However, in iterative development processes, these decisions may extend into later phases as the impact of the execution environment becomes more defined. While some design decisions may be deferred to later phases, such as deployment, configuration, or runtime, this deferral is a design decision. Consequently, even the initial decisions in software systems remain important and typically influence the overall architecture [JB05; MT10].

Architectural patterns provide a methodology for addressing recurring design challenges and organizing system components and their interactions. Common architectural patterns, including client-server, layered, event-driven, and microservice architectures, serve as templates that facilitate the structuring of the system. For example, a layered architecture allows for separating concerns by organizing components into layers with defined roles. In contrast, a microservice architecture enables scalability by allowing services to be deployed and scaled independently. However, this can increase the complexity of managing distributed systems [MT10; ZB12].

It is critical to document the software architecture. While code may reflect certain architectural elements, it is typically an incomplete representation because programming languages are not designed to capture architectural decisions. Therefore, software architectures must be explicitly documented using specialized languages and artifacts. This documentation facilitates the alignment between the initial requirements and the subsequent code, providing a framework for developing and maintaining the software. The software architecture provides a basis for stakeholder dialog, enables component reuse, and simplifies project management, planning, and cost estimation [Bac+00; JAV09].

Understanding component-based software engineering is critical to our view-based reverse engineering approach, which relies on identifying and reconstructing components from heterogeneous artifacts.

2.2. Component-Based Software Engineering

The concept of modularity is essential in software architecture. The decomposition process involves breaking down a complex system into discrete, manageable components. Each module encapsulates a specific portion of the system's functionality and can be developed, tested, and deployed independently, thereby promoting flexibility, reusability, and ease of maintenance. The relationships between these components, as defined by their interfaces, communication mechanisms, and data exchange protocols, are essential to the system's interaction model [Szy98; HC01].

Component-Based Software Engineering (CBSE) is a software development paradigm emphasizing building systems by integrating pre-existing, reusable components rather than developing all components from scratch. These components are self-contained units of functionality with well-defined, contractually specified interfaces that dictate how they interact with other components within the system. The modular nature of the components is an aspect of CBSE, as it allows them to be loosely coupled, increasing the flexibility and modularity of the system. This approach allows for the independent deployment, composition, and reuse of components, thereby enabling third parties to construct complex systems by assembling independently developed components [McI68; HC01].

Definition 3 (Software Component)

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." (Szyperski [Szy98])

McIlroy first introduced the concept of a software component [McI68]. Since then, the definition has evolved in Definition 3, resulting in many widely used implementations and frameworks [Szy98], including Enterprise Java Bean (EJB) [Ora13], Open Services Gateway initiative (OSGi) [The20], and the Common Object Request Broker Architecture (CORBA) [Obj21] component model. In the context of CBSE, a software component is defined as a compositional unit that exhibits explicit context dependencies and is characterized by contractually specified interfaces. These interfaces delineate the required interfaces and provide indispensable interfaces for the component to operate within a system. This concept is analogous to components in other engineering disciplines, where specified specifications and properties define the components in question [HC01].

An aspect of CBSE is the reuse of individual components, which are expected to be of higher quality due to the implementation of testing and validation processes. Component reuse can lead to cost savings in software development, especially when components are reused multiple times within or across systems. The composite pattern in CBSE enables the creation of components by integrating registration, authentication, and access control components into a user management system [FF95].

The CBSE development process differs from traditional procedural or object-oriented methodologies. In CBSE, the responsibilities of component developers, who are tasked

with creating individual components, and software architects, who are responsible for assembling these components into a complete application, are delineated. This distinction is reflected in development process models prioritizing building a functional system based on specified requirements. However, the CBSE approach presents several challenges, including supporting compatibility between components, especially when different teams have developed them or have undergone evolution. Versioning and dependency management are critical to maintaining system stability. Using third-party components can introduce various potential risks, including security, licensing, and long-term availability [Szy98; HC01].

To address these challenges, CBSE frameworks provide guidelines for best practices in component design, development, and composition. These frameworks facilitate the standardization of processes that enable organizations to reap the benefits of CBSE, including accelerated development cycles, reduced costs, and increased system reliability. Implementing CBSE requires a departure from traditional software development methodologies. It requires software engineers to consider component selection, evaluation, and assembly. By following these structured workflows, CBSE enables the creation of reliable, maintainable, and scalable software systems, thereby improving the efficiency and reliability of software development [Crn03].

It is essential to distinguish between system and component architecture in software architecture. System architecture is defined as the structure of the components that comprise a complete software system, including their respective responsibilities, interconnections, and the technologies used. In contrast, component architecture concerns application-level components and their interactions, abstracting away details such as user interfaces and data persistence for clarity. A component architecture is usually represented in UML component diagrams [UML; BR05]. Such a representation is logical and self-dependent on the technical implementation [HC01; Crn03].

2.2.1. Web Services

Based on the principles of CBSE, web services represent an application of components in distributed environments. Web services are a part of today's distributed computing, enabling communication and data exchange between systems over the Internet [Alo+04; Chi+07]. Web services are software systems that facilitate interoperable machine-to-machine interaction through standardized protocols, enabling seamless communication between applications regardless of the underlying platforms, languages, or technologies. The evolution of web services has been driven by the need for standardized methodologies to support the development of complex web applications. Adopting key technologies such as Hypertext Transfer Protocol (HTTP), Extensible Markup Language (XML), Simple Object Access Protocol (SOAP), and Web Services Description Language (WSDL) [Bra+08; Chi+07] has influenced the evolution of web services [Cur+02]. In the modern Internet-connected era, various services can be classified as web services. This thesis uses the definition provided by the World Wide Web Consortium (W3C) [HB04], the central standards organization for the World Wide Web, as outlined in Definition 4.

Definition 4 (Web Service)

"A web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format [...]. Other systems interact with the web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other web-related standards." (Haas et al. [HB04])

In a traditional client-server configuration, a client application sends a request to a server that oversees the web service, which then processes the request and returns an appropriate response. This interaction is typically accomplished using HTTP, the most common protocol for web communications. Web services' technologies include SOAP, Representational State Transfer (REST), and WSDL. SOAP is a protocol for exchanging structured information in web services that relies on XML for the message format and works over HTTP or other application layer protocols. It is platform-independent and language-agnostic, making it suitable for complex operations such as security, transactions, and asynchronous messaging [BDS08]. SOAP-based services are characterized by their adherence to standards and ability to accommodate functionality [Cur+02; Wan+04].

In contrast, REST represents an architectural style that uses standard HTTP methods, including GET, POST, PUT, and DELETE, to perform operations on resources identified by Uniform Resource Identifiers (URIs) [Ric06]. REST emphasizes simplicity, scalability, and statelessness, where each client request contains all the information the server needs to fulfill the request. REST-based services are typically easier to design, implement, and maintain than SOAP-based services. As a result, many developers have preferred REST-based services [Ric07; BB08].

Web services are essential to specific software development scenarios, particularly in Service-Oriented Architecture (SOA), where applications are built from loosely coupled services that communicate over a network [PL03]. The modularity of software components allows for flexibility and reusability, enabling organizations to better adapt to changing business needs. Web services are also a component of cloud computing, facilitating the delivery of Software as a Service (SaaS), Platform as a Service (PaaS), or Infrastructure as a Service (IaaS). Using standardized Application Programming Interfaces (APIs), cloud computing providers can offer scalable, on-demand resources to customers [Kav14].

While web services offer many benefits, they also present performance and scalability challenges. Services that require frequent data exchange or processing of large amounts of data can experience performance issues [BDS08]. Design and optimization support the service performs reliably and consistently across different environments and network conditions. Such techniques include load balancing, caching, and service replication [Alo+04; Wan+04].

2.2.2. Microservices

As a specialized form of component-based systems, microservices represent a new approach to software development characterized by decomposing large, monolithic applications into minor, independently deployable services. This architectural style improves scalability, resilience, and maintainability by facilitating the modularity and loose coupling critical to managing complex systems. Each microservice is associated with a discrete business function and operates as an isolated, self-contained unit that interacts with other services through well-defined interfaces, typically using lightweight communication protocols such as REST [FL14; Thö15].

Definition 5 (Microservice)

"The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, typically an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies." (Fowler et al. [FL14])

A principle of microservices is the concept of granularity, which means that each service is characterized by a high level of detail and a narrow focus on a specific business function. This facilitates the composability and reusability of services, allowing complex applications to be built from well-defined building blocks. Another principle is service autonomy, meaning each microservice can operate independently with its code base, data management, and deployment lifecycle. This autonomy enables agile development, testing, and deployment, reduces the risk of cascading failures, and increases the system's overall flexibility [Thö15; Dra+17].

Another defining characteristic of microservices is decentralization. This approach emphasizes that each service is responsible for independently managing its data, logic, and lifecycle, reducing dependencies, and enabling services to scale and evolve autonomously. Data management is decentralized in microservices, with each service maintaining its own database. This allows services to choose the most appropriate data storage technology and scale databases independently. However, this also creates challenges in maintaining data consistency, typically addressed through event-driven consistency models or architectures [Thö15; Dra+17].

Technological heterogeneity is a defining characteristic of microservices, allowing teams to select the most appropriate tools and frameworks for each service. This flexibility enables innovation but requires an assessment of potential interoperability and integration challenges. In addition, given the inherent probability of failure in distributed systems, supporting fault tolerance in the microservice context is critical. Techniques such as circuit

breakers, retries, and timeouts mitigate the impact of service failures, thereby supporting system resiliency and availability [Sil16; STV18].

However, the advent of microservices also introduces a level of complexity, particularly in the areas of service orchestration, management, and operational monitoring. The inherent difficulties of distributed systems, including those related to network latency, data consistency, and fault tolerance, become pronounced. Managing and monitoring multiple services across multiple platforms requires implementing orchestration tools and monitoring solutions with distributed tracing. To accommodate microservices, existing deployment and integration practices must be modified. CI/CD pipelines are critical to managing frequent updates and independent deployments, as these processes become prevalent in today's software development. Despite frequent changes, testing, deployment, and rollback process automation are required to maintain system stability and reliability [Dra+17; STV18; WLS20].

2.3. Software Weaknesses and Vulnerabilities

As software systems become more complex, their susceptibility to security breaches increases, making them attractive targets for malicious actors who exploit vulnerabilities to gain unauthorized access, disrupt operations, or execute malicious code. These vulnerabilities result from several factors, including the inherent complexity of software, suboptimal security practices during development, and the testing and quality assurance processes. While beneficial for accelerating growth, integrating third-party components introduces additional attack surfaces that must be managed, as flaws in these components can compromise the system's overall security [Ian+23; SDG23].

The cybersecurity industry relies on standardized systems such as Common Vulnerabilities and Exposures (CVE) [CVE] and Common Weakness Enumeration (CWE) [CWE] to address these challenges. The CVE system assigns identifiers to known vulnerabilities, enabling accurate referencing across the industry, which is essential for vulnerability management. This system allows for risk assessment, remediation prioritization, and rapid implementation of security patches. In contrast, the CWE is concerned with classifying software weaknesses that can potentially lead to vulnerabilities. It categorizes issues and describes these vulnerabilities, including their potential impact and mitigation strategies. The CWE framework provides an overview of common types of vulnerabilities. At the same time, the CVE system identifies specific instances of these vulnerabilities in software products. For example, CVE-2022-22965 [US 23; Sny22] is associated with a remote code execution vulnerability in the Spring framework [VMw24; VMw22]. The CVE system is further expanded by the U.S. National Vulnerability Database (NVD) [US 24], maintained by the National Institute of Standards and Technology (NIST). The NVD serves as a repository of information about software vulnerabilities, including metadata about the severity of the vulnerability and the components affected [Mar19; GGB23; Eld+24].

Integrating these classification systems with vulnerability scoring systems, such as the Common Vulnerability Scoring System (CVSS) [CVSS], improves the ability to assess vulnerabilities' severity and potential impact. The CVSS takes a quantitative approach to determining the severity of vulnerabilities by considering several factors, including the attack vector and its implications for confidentiality, integrity, and availability. This rating system requires that published ratings include the associated rating vector, which provides transparency and facilitates the reuse of individual metrics in different contexts [Mar19; Eld+24].

In addition, it is critical to understand the processes used to exploit vulnerabilities. Vulnerabilities can be classified according to their attack vectors, including those exploited via network connections and those requiring local access to a system. For example, network-based vulnerabilities include insecure network configurations and unpatched open ports. In contrast, local vulnerabilities include flaws that allow users to gain unauthorized administrative privileges through privilege escalation [WHR22; Wal24; WHR23].

With the spread of cybersecurity, organizations are turning to platforms like Snyk [Sny24] that integrate security tools directly into the development lifecycle [Sus+23]. By leveraging standardized systems such as the CVE and the CWE system, these platforms facilitate identifying and remedying vulnerabilities in software development. Such methodologies promote a proactive approach to security, in line with the increasing emphasis on integrating security considerations throughout the software development lifecycle [Ian+23; SM23].

2.4. Model-Driven Software Development

Model-Driven Software Development (MDSD) is an approach to software engineering that redefines the role of models. Rather than being viewed as mere documentation, models are integral to the development process. In contrast to traditional code-centric methodologies, MDSD prioritizes creating and manipulating models that encapsulate various aspects of software, including structure, behavior, and data flow, at different levels of abstraction [Béz05]. Such models are considered artifacts that guide the entire software development lifecycle. They also serve as high-level conceptual tools, facilitating stakeholder communication and collaboration by providing a common lexicon for understanding the system [CFM03; Völ13].

Definition 6 (Model)

"A formal representation of entities and relations in the real world (abstraction) with a certain correspondence (isomorphism) for a particular purpose (pragmatics)." (Stachowiak [Sta73])

The theoretical foundation of MDSD is rooted in model theory, particularly the work of Stachowiak, who identifies representation, abstraction, and pragmatism as the properties

of a model within Definition 6 [Sta73]. The representation aspect of MDSD refers to the ability of a model to represent a particular entity in a manner that is faithful to its inherent properties. In contrast, the abstraction process involves the reduction of complexity by identifying and emphasizing relevant aspects. The pragmatism of a model can be defined as its function within the development process [Béz05]. The foundation is extended by conformance, which emphasizes the alignment between a model and its corresponding metamodel, as defined in Definition 7. This concept differs from the traditional object-oriented approach to instantiation [Sta73; Völ13].

Definition 7 (Metamodel)

"Metamodels are models that make statements about modeling. They describe the possible structure of models and define the constructs of a modeling language and their relationships, as well as constraints and modeling rules." (Völter [Völ13])

One principle of MDSD is abstraction, which allows software systems to be represented at different levels of detail. This will enable developers to focus on creating a design and defining the system's functionality rather than being constrained by the specifics of the implementation process. MDSD models can be UML diagrams [UML; BR05], Domain-Specific Languages (DSLs), or platform-independent specifications, each tailored to specific requirements and domains. DSLs are advantageous in this context because they are designed for domain experts, thereby increasing the accuracy and relevance of the models [Béz05; Völ13; Voe+13].

An understanding of MDSD is essential to our view-based approach because it uses models to represent software systems and their components. Applying the principles of MDSD makes creating abstractions that facilitate the reverse engineering process possible.

2.4.1. Meta Object Facility

Meta Object Facility (MOF), standardized by the Object Management Group (OMG), is a framework for managing metadata and enabling model-driven system developments, particularly in software engineering. The MOF is based on a four-level architectural framework that includes levels M0 through M3. The M3 level represents the meta-metamodel, which is self-descriptive and defines the structure and semantics for metamodels at the M2 level. The M2 level includes metamodels, such as the UML, representing models at the M1 level. The M1 models, in turn, describe the structure and behavior of runtime data and system instances at the M0 level. This hierarchical and recursive architecture guarantees that each level adheres to the rules and structures defined by the level above it, thereby supporting an abstraction system. These levels are visually represented in Figure 2.1.

MOF's self-referential nature and genericity allow it to be used across different domain-specific languages, making it an indispensable component of Model-Driven Engineering (MDE) methodologies such as MDSD. MOF offers two variants: the Complete Meta Object

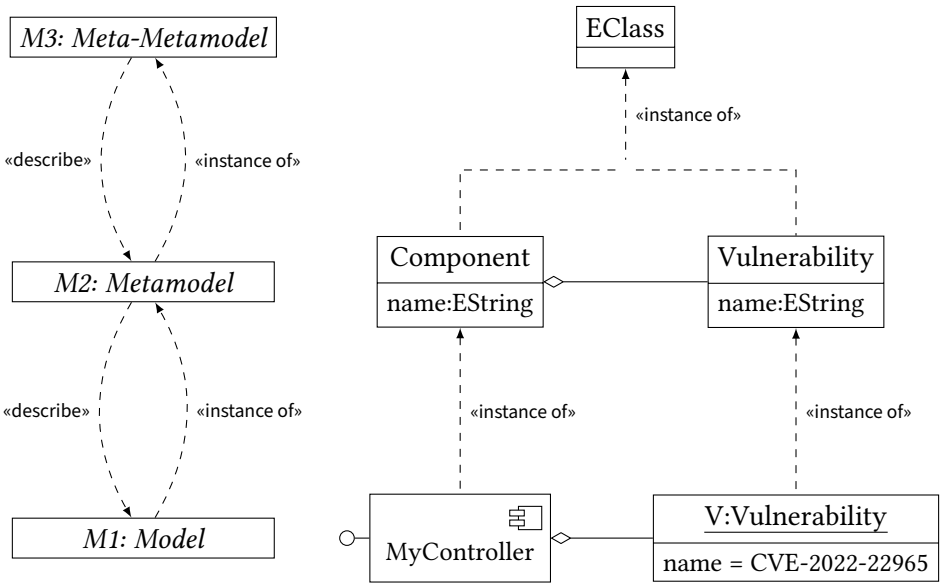


Figure 2.1.: Example of the MOF framework for representing component-based software systems. The framework starts with M3, the meta-metamodel, which provides the highest level of abstraction. It continues with M2, the metamodel, which describes the structure of models. Finally, M1 is the model in which specific instances, such as software components and vulnerabilities, are directly represented. This diagram is derived from [UML].

Facility (CMOF) is designed to address complex metamodeling needs. At the same time, the Essential Meta Object Facility (EMOF) provides a simplified structure derived from UML class models. EMOF, which adheres to the same four-layer architecture, is implemented within the EMF via Ecore [Ecl24a; Ecl24c]. EMF's compatibility and implementation flexibility make it a valuable tool for mapping MOF models to systems and supporting interoperability in model-driven development.

The key metamodeling constructs provided by MOF include classes, attributes, operations, associations, and generalizations, all of which are analogous to the elements of object-oriented programming. These constructs facilitate the creation of complex, modular metamodels for defining DSLs. The standardized structure of MOF promotes interoperability, reusability, and tool integration, which is necessary for model-driven approaches, where models must be transferred and transformed across systems and platforms.

2.4.2. Eclipse Modeling Framework

The EMF [Ecl24c] is a Java-based, open-source toolkit integrated into the Eclipse platform [Ecl24a] that plays a central role in model-driven software development [Ste11]. At the core of the EMF is the Ecore metamodel, which serves as a metamodeling framework analogous to the OMG's MOF. It implements the EMOF standard. The Ecore metamodel provides a set of constructs, including `EClass`, `EAttribute`, `EReference`, and `EOperation`, that facilitate defining and manipulating domain-specific models [Ecl24c]. These constructs allow developers to define classes, attributes, references, and operations within their

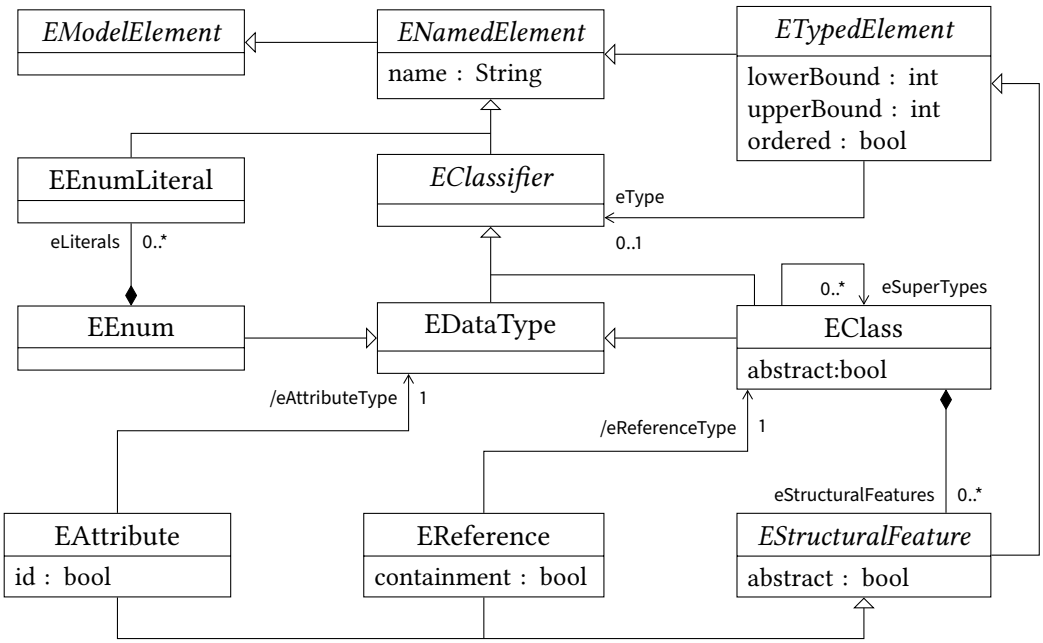


Figure 2.2.: The simplified class diagram overviews the metaclasses within the Ecore modeling formalism. The original diagram [Ste11; Bur14] is used here in the adapted version [Kra19; Kla22].

models. A feature of Ecore is the distinction between two types of properties: attributes, which refer to basic data types and enumerations, and references, which point to other classes [Ste11]. Figure 2.2 visually represents the elements of the Ecore metamodel, presented as a UML class diagram [Ste11; Bur14; Kra19; Kla22].

EMF's code generation capabilities are critical because they facilitate the automatic generation of Java classes from Ecore models. This includes generating model classes, adapters, factories, and notification mechanisms. This automation reduces the need for manual coding and supports consistency between the model and its implementation. It also increases productivity by allowing developers to focus on defining the structure and behavior of the model [Ecl24c; Ste11]. In addition, the framework enables the creation of DSLs through tools such as Xtext, which defines the grammar of a language, derives the metamodel, and generates parsers and editors [Bet16].

The EMF runtime environment includes a set of APIs and tools for managing model instances. These capabilities include loading, saving, validating, and serializing data. EMF's capabilities include validating, querying, and transforming models using Query/View/-Transformation (QVT) [QVT] or Atlas Transformation Language (ATL) [Ecl23; JK06]. In addition, the EMF Compare tool provides the ability to compare and merge different versions of models, which is useful in collaborative contexts. The framework's support for constructs such as inheritance, composition, and containment further improves its flexibility in managing complex data structures [Ecl24c; Ste11].

2.4.3. Model Transformation Languages

Model Transformation Languages (MTLs) are indispensable tools within MDSD for defining and executing transformations between different models or from models to textual artifacts such as source code. Definition 8 suggests that transformations can be considered the "heart and soul" of MDSD [SK03]. The metamodel concept serves as the foundation for MTLs. It provides the abstract syntax and semantics for input and output models, supporting the consistency and correctness of transformations [Völ13].

Definition 8 (Model Transformation)
"Automated processes that take one or more source models as input and produce one or more target models as output, while following a set of transformation rules." (Sendall et al. [SK03])

Model-to-Model (M2M) and Text-to-Model (T2M) transformations are operations supported by MTLs and are essential for various software development tasks. In an M2M transformation, a source model is transformed into a target model, and both models must adhere to the constraints of their respective metamodels [HKT22]. Figure 2.3 is a graphic representation of the described concept and provides a visual aid to understanding [Völ13; Koz08]. This type of transformation is used for tasks such as mapping a UML class diagram [UML; BR05] to an alternative model type. In contrast, T2M transformations generate models from textual inputs, including source code, configuration files, or documentation derived from a model [Höp+22]. This is critical for bridging the gap between abstract models and executable systems. While languages such as Xtend [Ecl24d] are used for T2M transformations, they also facilitate the generation of source code and other textual artifacts through template-based or imperative approaches [Bet16]. Xtend is designed to integrate seamlessly with Java, combining generated and manually written code [Heb+18].

Implementing these transformations is based on rules that specify how elements present in the source model are mapped to elements present in the target model. The transformation engines execute these rules, automating the process and supporting metamodel compliance. MTLs can be classified according to their formalisms and the types of transformations they perform [HKT22]. For example, rule-based MTLs such as QVT [QVT] and ATL [Ecl23; JK06] use declarative and imperative rules to manage M2M transformations. ATL can support declarative and imperative constructs and is integrated with the EMF [Heb+18; Ste11].

2.4.4. View-Based Modelling

View-based and multi-view modeling are software engineering methodologies representing a system through multiple complementary views, each focusing on specific aspects or concerns relevant to different stakeholders or stages of the development process. Following the principle of separation of concerns, these approaches facilitate the targeted analysis

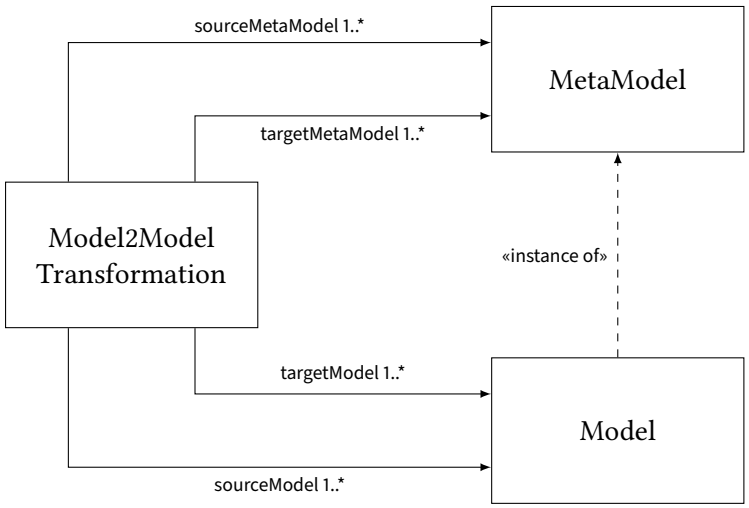


Figure 2.3.: A model-to-model transformation is a mapping process in which source metamodel elements are mapped to target metamodel elements to create a new model derived from a source model. Koziolek [Koz08] used the original diagram from Völter [Völ13] as a reference for this diagram.

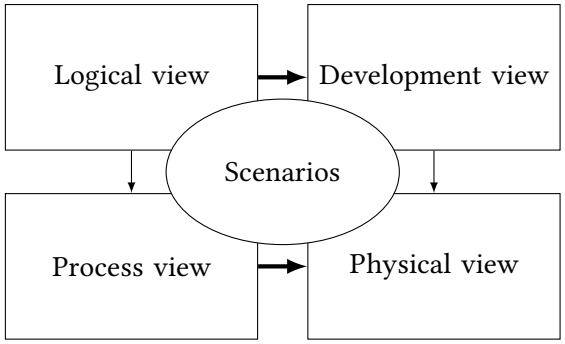


Figure 2.4.: The 4+1 View Model describes the architectural configuration of software-intensive systems. The model is based on integrating multiple concurrent views, including logical, developmental, process, and physical. In addition, selected use cases or scenarios illustrate the architectural design and function as an additional view to provide a diagram of the system's functionality [Kru95].

of various facets of the system, thereby addressing the inherent complexity of large systems [WAA85]. In software architecture, stakeholders require a narrower focus on specific system subsets. This, in turn, requires methodologies that can either streamline or obfuscate the system's inherent complexity. This selective focus allows stakeholders to address only those elements relevant to their respective roles and prevents them from being overwhelmed by the system details [Fin+92].

The approach of view-based development separates concerns by providing stakeholders with information tailored to their specific needs. This is situated within the field of architectural description languages, which play a role in navigating the intricacies associated with large software systems [RT14]. The construction of a system into multiple views or viewpoints allows for the isolation of different aspects of the system, thereby facilitating the analysis of each element in isolation. Philippe Kruchten's "4+1 View Model" (see Figure 2.4) deserves special mention among the methodologies used in view-based

development [Kru95]. This model provides a structured approach to organizing the views associated with a software system and categorizes views into four categories: logical, development, process, and physical. It also includes scenarios that describe use cases or interactions. This framework accommodates the various stakeholders' interests and technical backgrounds, facilitating a collaborative approach to the development process [Kru95].

Definition 9 (View)

"A view is a separately identifiable body of information created, stored, and delivered for human and machine use (the information part) and comprises a portion of a work product that expresses a software architecture (the architecture description)." (International Organization for Standardization [I42010])

A view represents a system from a particular viewpoint, such as its functionality, performance, or security. In contrast, a viewpoint defines the conventions and rules for constructing those views. The dual concept of views and viewpoints, as defined by the ISO/IEC/IEEE 42010 standard [I42010], provides a structured approach to creating architectural descriptions that address the concerns of various stakeholders. This standard is an international benchmark for architectural description and provides guidelines for articulating the relationships between models and views within system architectures [I42010]. Following Definition 9 in this standard, a view addresses specific stakeholder concerns. At the same time, a model provides an abstract representation of the system's architectural components, interconnections, and attributes. Each view is derived from the model by applying a specific view type, which serves as a metamodel for the view [GBB12]. This view type is defined in Definition 10 as the permissible elements and relationships the view can contain, thereby structuring the view according to predetermined standards and requirements. Consequently, views serve as a mechanism for abstracting from the completeness of the modeled information, thereby simplifying the system's inherent complexity [Bur14; Bur+14].

Definition 10 (View Type)

"A view type is a set of conventions for creating, interpreting, and using an architecture view to frame one or more matters relevant or important to a stakeholder (the concerns)." (International Organization for Standardization [I42010])

The standard elaborates on two architectural views: projective and synthetic [I42010]. Projective views are derived directly from an underlying model and present a selective view that emphasizes certain aspects of the model based on the interests of relevant stakeholders. These views allow stakeholders to analyze the model from various viewpoints, which can be critical for addressing functional and non-functional requirements. In synthetic approaches, views are manually constructed from system descriptions, requiring consistency management [Bur14; Bur+14]. As the number of transformations between views

increases, managing these transformations becomes increasingly complex. In contrast, the projective approach derives views from a Single Underlying Model (SUM), allowing for efficient consistency management due to the centralized nature of the underlying model [ASB10].

2.5. Architecture Description Languages

An Architecture Description Language (ADL) is defined in Definition 11 as a formal, expressive language that combines specified syntax and semantics to describe the architecture of an entity [I42010]. It also serves as a structured communication tool between stakeholders involved in the architectural process. ADLs are used in various disciplines, including systems engineering, software engineering, and enterprise modeling. They facilitate the representation of architectural structures, enabling the creation and understanding of architectural views [I42010]. This is achieved by articulating architectural considerations using specific architectural description elements relevant to the entity of interest and the architectural context. ADLs represent architectural components, including components, connectors, and configurations. This supports that complex systems can be communicated with clarity and precision [MT00].

Definition 11 (Architecture Description Language)

"Means of expression, with syntax and semantics, consisting of a set of representations, conventions, and associated rules intended to be used to describe an architecture." (International Organization for Standardization [I42010])

ADLs provide a conceptual model for representing system architectures, enabling improved communication and analysis. In software engineering, computer languages describe software architectures by representing the interactions and configurations of components and connectors. This structured approach enables preliminary validation and feasibility testing of designs and facilitates adherence to design principles and non-functional requirements such as performance and security [Cle96]. Selecting an appropriate ADL requires considering view methods, which specify how information is selected, transformed, and presented in architectural views [I42010]. These methods determine the information required to capture, analyze, and describe architectural concepts and features, contributing to the development process. The formality of an ADL's semantics can vary widely, from natural language glossaries to complex ontological theories using formal logic [Cle96].

The evolution of ADLs has progressed from informal representations, such as annotated box-and-line drawings, to formal approaches that address the limitations of earlier methods. This evolution has facilitated two-way communication, embedded initial design decisions, and improved the overall system design and analysis process [Pan10]. Despite the challenges associated with their adoption, such as learning curves and tool support, ADLs are critical in domains where clarity and early analysis. Ensuring consistency, or traceability,

between different architectural description elements that refer to the same domain entity is essential. This is achieved by establishing correspondences or implementing a unified underlying ontology that integrates the constructs of the ADLs in use [MT00; Pan10].

The following two ADLs are particularly relevant to our approach because they provide a standardized and expressive means of representing software systems. These ADLs are used to model the reverse-engineered system and its components, thereby facilitating the analysis and evaluation of the system's architecture. Although our approach focuses on these two ADLs, the approach presented in this thesis can be applied to other ADLs.

2.5.1. Unified Modeling Language

The UML is a standardized, general-purpose ADL that plays a role in software engineering [UML]. It is used to visualize, design, document, and construct software and other systems [Med+02]. UML provides a set of diagrams that can be divided into structure and behavior. Structure diagrams, including class, object, component, and deployment diagrams, represent the static aspects of a system. These diagrams focus on the architectural structure of the system, its organizational configuration, and the relationships between its constituent elements [Med+02]. These diagrams are valuable for understanding and documenting how a system is partitioned into components, dependencies, and physical deployment on hardware nodes. Behavioral diagrams, including use case, sequence, activity, and state machine diagrams, illustrate the dynamic aspects of a system. Such diagrams document functionality, processes, and user interactions over time. Interaction diagrams, including sequence diagrams, show the control and data flow between system components [UML].

Definition 12 (Model Element)

"A model element is a constituent of a model. A named element is an element in a model that may have a name. A relationship is an element that specifies some kind of relationship between other elements." (Object Management Group (OMG) [UML])

The elements of UML models are model elements with relationships, which serve as the basic units of these diagrams. During execution, the structural and behavioral aspects of the system must be represented. This can be achieved by focusing on the constituent components and their interactions. According to Definition 13, the artifacts in UML models, representing information such as model files, source code, or other development artifacts, are critical to understanding and documenting the system architecture [UML; Med+02].

Definition 13 (Artifact)

"An artifact is the specification of a physical piece of information that is used or produced by a software development process, or by deployment and operation of a system." (Object Management Group (OMG) [UML])

Viewpoint	View Type	
	System-Specific	System-Independent
Structural	Assembly	Repository
Behavioral	Usage model	SEFF
Deployment	Allocation	Resource environment
Attack	Access Control	Vulnerabilities

Table 2.1.: The PCM is organized around several viewpoints. This table presents data derived from [Reu+16] and supplemented by the last line of [Wal24].

Creating UML diagrams is facilitated by specialized software, including PlantUML and open-source tools. Such tools enable the creation of different types of diagrams, thereby increasing the efficiency of the modeling process. Despite the standardization of UML and its advantages in object-oriented design and system architecture, many software engineers continue to use informal, hand-drawn diagrams. Such diagrams typically incorporate elements of UML, but may not strictly adhere to its standards. This represents a balance between formal rigor and practical in software development [Roq+24; Hal+24].

2.5.2. Palladio Component Model

The PCM is a DSL and ADL designed to facilitate the specification, documentation, and analysis of component-based software architectures [Bec08; Bec10; Reu+11; Reu+16]. The PCM enables predicting and evaluating software performance and reliability during the software life cycle [BKR07; BKR09; HKR11; KR08a; Koz13; KR08b; Mar+10] and attack propagation analysis [WHR22; WHR23; Wal24] techniques. These analysis techniques facilitate the identification of potential bottlenecks, attack vectors, and reliability issues, enabling informed decision-making before system implementation. PCM is organized around several viewpoints listed in Table 2.1. Each viewpoint consists of specific view types that capture particular aspects of a software system.

The structural viewpoint addresses static properties and distinguishes between system-specific and system-independent types. The repository view type, which is system-independent, encompasses components and interfaces that can be reused across multiple systems. In contrast, the assembly viewpoint describes how these components are instantiated and interconnected within a specific system. In addition, the structural viewpoint emphasizes the dependency structure of systems and the definition of their basic components. It is important to solicit input from key individuals, such as the system architect and component developer, and to emphasize their indispensable role in the design process [Reu+16].

The behavioral viewpoint is concerned with both functional and extra-functional execution semantics. The intra-component behavioral viewpoint concerns the internal operations of individual components that operate in isolation. In contrast, the inter-component behavioral viewpoint emphasizes the interactions between components. In addition, the

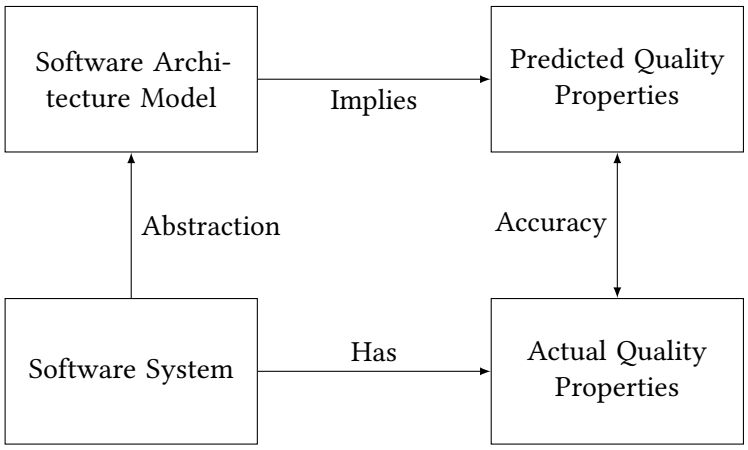


Figure 2.5.: Model-Driven Quality Prediction (MDQP) is demonstrated through the interrelationship between a software architecture model and the predicted quality characteristics and their correlation with the actual quality characteristics observed in the software system. This provides insight into the process of abstraction from software systems to architecture models and the assessment of accuracy in predicting and actualizing quality characteristics. This diagram is derived from [Koz13].

usage model viewpoint describes the behavior of users or external systems interacting with the software. This is an aspect of understanding usage patterns that affect system performance [Reu+16].

The deployment viewpoint in PCM addresses assigning component instances to physical or virtual resources within the execution environment. This viewpoint includes two additional view types: the allocation view type, which describes the assignment of component instances to specific resource containers, and the resource environment view type, which models the hardware infrastructure and network connections, including processing nodes and interconnecting resources. PCM treats physical and virtual resources equally, although the specific characteristics of virtual resources, such as changing performance characteristics, remain a topic of ongoing research. The deployment viewpoint is concerned with specifying the execution environment and supporting that software components are appropriately allocated to available resources [Reu+16].

The PCM can be used to identify and predict the quality attributes of a software system. This process is called MDQP. MDQP can predict quality attributes associated with various points of interest within a software system. In the context of MDQP, the software system may already be implemented or exist only as a design. The quality attributes manifest only when the system design is realized in an implementation. The quality characteristics of a software system depend on both the design and the decisions made during implementation. This MDQP concept is illustrated in Figure 2.5. The PCM is central to predicting and analyzing non-functional qualities (such as security and performance) in component-based software systems. Palladio provides an approach for evaluating design alternatives and making informed decisions about quality attributes. Its ability to link quality attributes across components makes it particularly well-suited for assessing models generated by Model-Driven Reverse Engineering (MDRE). As a component-based approach, Palladio

is consistent with the principles of CBSE, as evidenced by the case studies and examples presented in the following sections [Mar+10; Koz13].

Walter aims to improve system confidentiality by addressing the complexity of access control and the vulnerabilities that arise from the increasing digitization of systems. To address these challenges, the authors introduce two different metamodels. The first metamodel is explicitly designed for context-based access control policies. This allows for the precise delineation of the circumstances under which access is permitted, thereby facilitating the modification of access privileges by evolving contextual variables, such as time shifts or user location. This model provides a system-specific view for analyzing potential attacks. It is closely tied to a given system's configurations and requirements and modifies access control measures based on system-specific contexts and scenarios. The second metamodel is concerned with the identification and analysis of system vulnerabilities. It provides a representation of potential vulnerabilities and the requirements for exploitation, considering both authorization levels and the context of the attacker. This vulnerability model provides a system-independent view of attack modeling. The model identifies vulnerabilities that can be exploited regardless of the specific system architecture. Walter offers a generalized approach that can be applied across systems to identify potential attack paths, including lateral movement and vulnerability chaining. These models aim to improve system security by facilitating analysis of access control policies and vulnerabilities [WHR22; WHR23; Wal24].

The abstraction mechanism of PCM allows the generation of performance or attack propagation simulation models derived from various sub-models, including those representing static architecture, component behavior, resource environment, usage profile, and allocation. These models facilitate the evaluation of architectural decisions, the optimization of resource allocation, and the resolution of performance issues before system implementation. This supports the analysis, which is scalable and precise without sacrificing accuracy [RK11; Bro+12].

2.5.2.1. Structural Viewpoint

The PCM focuses on a system's architectural components, interfaces, and connectors. In a PCM repository, interfaces and components are considered entities that exist independently and are not inherently bound to other entities. Interaction between components and interfaces is facilitated by the exposure or invocation of these interfaces, as indicated by the associated exposed or invoked role. Each interface is associated with a set of service signatures defining the interface's services. The parameters in these signatures are characterized to highlight relevant properties, such as the size of a list or specific data type properties [Reu+11; Reu+16].

In the context of the PCM, components are conceptualized as black-box entities, defined by their relationship to interfaces rather than by the presence of those interfaces within them. The separation of interfaces within repositories emphasizes their independence, allowing software architects to construct systems that satisfy particular requirements by

combining independently developed components and interfaces. A component assigned a specific role regarding an interface is responsible for implementing the services associated with that interface. In contrast, elements required to fulfill a particular role depend on external services, the importance of which varies by component type [Bec10; Reu+16].

In the context of PCM, connectors facilitate the interaction between components by linking their respective roles. The different types of connectors link these roles, allowing multiple components to share the same interfaces and forming complex interaction patterns, such as chains of responsibility. PCM distinguishes between different types of components, in particular between basic and composite components. Basic components implement their services through Service Effect Specifications (SEFFs), whereas composite components comprise subcomponents. The SEFFs inherent in component specifications are included in PCM repositories but are not considered top-level entities due to their dependence on specific components [Bec10; Reu+16].

PCM's approach to defining interfaces and data types is essential to modeling component behavior. The definition of interfaces includes the specification of service signatures and, in some cases, the inclusion of a protocol that constrains valid call sequences. Signatures are ordered parameters corresponding to specific data types and classified as input, output, or input/output parameters. PCM data types fall into three categories: primitive, collection, and composite. Each data type serves a specific purpose in the context of component interactions [Bec10; Reu+16].

2.5.2.2. Behavioral Viewpoint

SEFFs in the PCM are a tool for analysis of the behavior of services provided by software components. This is critical for making accurate performance and reliability assessments in component-based systems. SEFFs define the control and data flow of a component's service behavior, including interactions with other components, by specifying the order of required service calls, their parameters, and the conditions under which they are executed. This delineation allows SEFFs to facilitate the integration of high-level architectural design with the behavior of individual components. This supports the idea that performance, reliability, and resource utilization are incorporated into the development process [RK09].

SEFFs use various actions to model service behavior. An external call represents an interaction in which a service provided by one component calls a service provided by another component. Such interactions can be used to model dependencies between components, making it easier to analyze the impact of these dependencies on overall system performance. Loop actions simulate repetitive control flows, which is essential for elucidating the influence of iterative processes on resource consumption. Branch actions introduce decision points based on input parameters or probabilities, allowing complex, conditional behavior to be modeled within a service's control flow. Internal actions abstract the component's internal operations, such as computational tasks or state changes, that consume CPU time or memory [Bec10; Reu+16].

An advantage of SEFFs is their ability to abstract from source code, encompassing behavior across multiple classes and methods while facilitating modular performance analysis. SEFFs define resource requirements in a general, abstract way without referencing specific timing values. This increases the reusability and adaptability of specifications. The parametric nature of SEFFs allows for modeling performance-influencing factors, including the characteristics of the required services, the deployment platforms, and the varying service parameters. This will enable SEFFs to adapt to changing usage conditions. This adaptability is particularly beneficial because it allows SEFFs to adjust based on different input parameter characterizations that affect resource requirements, branching conditions, and loop iteration counts [KBH07; Koz08].

Although SEFFs do not directly model a component's internal state, they provide mechanisms for referencing and exploiting component parameters that may reflect that internal state, thus enabling the specification of service behavior. Storing and reusing SEFFs in repositories further enhances their utility by allowing software architects to assemble them into complex architectures. SEFFs facilitate the prediction of system performance, the analysis of data and control flow, and the evaluation of alternative architectural options in different contexts and usage scenarios [KBH07; Koz08].

2.5.2.3. Deployment Viewpoint

In the PCM, accurately representing the resource environment is essential to predicting system quality because resources impact application responsiveness and throughput. The PCM accomplishes this by assembling software components and modeling the resource environment, including processing and linking resources. Processing resources are encapsulated in resource containers, such as servers or virtual machines, on which the components run. These containers are defined by attributes, including processing rates, scheduling policies, and the types of resources they contain, such as processing units and memory. In contrast, the linking resources represent the network connections between these containers and are characterized by attributes such as latency and throughput [KKR11; Reu+16].

The PCM provides a conceptual framework for representing resources that allows developers to refer to generalized categories of resources, without specifying their precise attributes. This abstraction is essential to address the deployment scenarios that require independent quality specifications, as component developers typically define these specifications without regard to specific hardware configurations. These resource categories are stored in resource repositories, supporting consistency across component specifications and facilitating reuse in different contexts [KR08b; HKR11].

Equally important is the deployment model, which defines how software components are deployed to hardware resources. The mapping context defines the relationship between software components and resource containers, thereby determining how the system responds to resource requests generated by components. This mapping is critical for analyzing system performance and supporting the deployment to meet quality requirements.

In addition, the mapping model considers network resources, describing how communication between components uses the network connections between different resource containers. By modeling these aspects, PCM enables software architects to anticipate system quality, identify potential bottlenecks, and optimize deployment to meet resource constraints and quality goals [BKR07; BKR09].

2.5.2.4. Vulnerability Viewpoint

Walter aims to improve system confidentiality by developing models that address access control and software architecture vulnerabilities. As digitization increases, the specification of access control policies becomes complex, requiring the incorporation of contextual variables such as user location or time, which affect the overall impact of the policy. To address this complexity, Walter has developed a metamodel explicitly designed to specify context-based access control policies. The metamodel allows for precise delineation of the circumstances under which access is permitted, facilitating the adaptation of access privileges to fluctuating contextual data. Vulnerabilities threaten system confidentiality, as they can allow unauthorized access to protected entities and result in the disclosure of sensitive information. To address these issues, Walter developed a vulnerability metamodel that represents potential system weaknesses and their exploitation conditions, considering both the required authorization level and the context of the attacker. The model identifies attack paths involving lateral movement and vulnerability chaining, which is important for understanding how multiple security breaches can occur sequentially within a system [WHR22; WHR23; Wal24].

Access control policies are essential to maintaining confidentiality, as they regulate access to architectural elements and protect against unauthorized use. It is important to note that access control is not the only protection mechanism. However, it remains a protection method, along with other mechanisms such as usage control and encryption. To facilitate the expression and evaluation of access control policies, Walter developed a metamodel that allows architects to define and analyze these policies. This model uses the System Entity attribute and associates attributes with entities within the PCM, a superclass for various architectural elements. This flexibility is not limited to a single architectural layer but extends to associating attributes with other components, including assembly contexts and resource containers. This model allows for a definition of attributes, increasing the adaptability of access control policies to the specific context required for access without requiring changes to the metamodel. This approach uses attribute-based access control and incorporates the eXtensible Access Control Markup Language (XACML). XACML's ability to model multiple data types and its status as a standardized format allow architects to integrate custom XACML-based statements directly into their designs. However, it is the architect's responsibility to monitor the specifics of the PCM integration to ensure that the attributes of the designated entities are output correctly to prevent security breaches [WHR22; WHR23; Wal24].

Walter extends the PCM by introducing a vulnerability property through a metamodel extension. The integration manages the relationship between PCM elements and the

architectural framework and supports multiple architectural elements, including composite components, basic components, and resource containers. Walter emphasizes the importance of modeling vulnerabilities and characterizing their associated properties, which is necessary for accurate security risk assessment. This approach provides an understanding of identified and unidentified vulnerabilities within systems, which is critical for identifying potential security threats through models such as threat modeling. Walter developed a dedicated metamodel to address vulnerabilities. This was achieved by integrating knowledge from the CWE, the CVE, and the CVSS. The metamodel distinguishes between CVE vulnerabilities, based on specific CVE identifiers, and CWE vulnerabilities, based on categories. Both types are derived from a parent class that encapsulates CWE references, demonstrating the adaptability of this classification to multiple CVEs. The metamodel describes vulnerabilities using attributes that elucidate their exploitability and impact, thereby avoiding reliance on CVSS ratings. In contrast, the attributes are aligned with those used in NVD and incident reports, facilitating integration and comparison [WHR22; WHR23; Wal24].

Combining these metamodels allows for security analysis during the software life cycle phase based on the principles of security by design. This approach enables accurate identification of access violations and compromised elements while improving the ability to assess the impact of security measures within the architectural framework. Despite scalability limitations, the models proposed by Walter are effective in smaller architectural contexts and hold promise for reducing effort and improving security outcomes in software development [WHR22; WHR23; Wal24].

2.6. Reverse Engineering

Reverse engineering deconstructs a system, software, or device to reveal its underlying design, functionality, and operating mechanics [Rek85]. This is accomplished by breaking down the system into its basic components. This process is essential for extracting knowledge or design data from an existing product that can be used to reconstruct, improve, or modify it. In software engineering, reverse engineering examines a software system's code base, architectural framework, and runtime behavior. This analysis is appropriate when inadequate documentation or source code prevents insight into the software system. This highlights the role of reverse engineering in understanding and improving software architectures, strengthening system security, and supporting system integration initiatives [CS10].

In software architecture, reverse engineering is a process that involves deconstructing the code base and examining the runtime behavior of the system. This enables the inference and delineation of the architectural structure, design patterns, and other high-level abstractions that comprise the system's overall design. This practice is advantageous for systems that lack documentation or have undergone modifications that may obscure the original architectural design [MWT95; Lun08]. An approach in this area is MDRE,

which integrates model-driven software development methodologies to improve the understanding and exploration of software systems. Unlike software reverse engineering, which is concerned with code-level details, MDRE focuses on abstracting key structural and behavioral attributes into models. Such models represent the system, enabling the identification of underlying patterns and structures [RS04].

2.6.1. Software Architecture Reverse Engineering

The reverse engineering software architecture process is critical for extracting and reconstructing a software system's architectural structure, design patterns, and high-level abstractions from its existing code base and runtime behavior. This process applies to systems with inadequate or missing documentation or systems where the original architectural design has been modified. The goal is to recover the architectural blueprints to improve understanding of the system, facilitate maintenance, and support evolution, thereby supporting long-term maintainability and ongoing development [MWT95].

The reverse engineering process begins with collecting key artifacts, including source code, build scripts, runtime logs, and accessible documentation. These serve as the basis for the subsequent analysis. The following analysis phase is divided into two distinct methods: static and dynamic analysis. Static analysis is a method of examining source code or compiled binaries without running the system. The goal is to analyze the system's dependencies, identify individual modules and packages, and find patterns in the data. This approach provides insight into the structure of classes, the logical dependencies between them, and the overall system architecture [CS10]. However, dynamic analysis complements the system because static analysis may not capture the system's runtime behavior. Dynamic analysis involves observing the system's behavior during execution, capturing real-time interactions, control flow, and performance metrics. This is useful for finding runtime bindings and behaviors that may be obscured by static analysis. However, this method is more time-consuming than static analysis because of the need to set up an appropriate test environment. It allows the examination of specific software system components executed during the test phase [Lun08].

The system architecture is then reconstructed by identifying its components, analyzing their relationships, and visualizing their structure. The typical output of such tools is a visual representation, such as a class diagram or a dependency graph, which facilitates understanding the reconstructed architecture. However, these derived models typically require manual review and refinement to resolve ambiguities and support completeness, as the mapping between the architectural model and the source code is not always explicit. Such a mapping is sometimes implicit and exists only in the mind of the software developer or architect [Str+06].

It is imperative to acknowledge the inherent difficulties of reverse engineering. The complexity and size of large systems, especially those with high concurrency or distributed components, can make the process both resource-intensive and challenging. A process that takes more than source code as input is necessary to achieve accurate results because

an architectural model contains elements that are not present in the source code alone. In addition, architectural erosion, defined as the deviation of a system from its intended design over time, adds a layer of complexity to the reconstruction process [CD07]. While tools can automate many tasks, they may not capture the architecture, resulting in incomplete or ambiguous results. As a result, creating an accurate architectural model sometimes requires manual interpretation and refinement [Mül+00].

Despite these challenges, reverse engineering remains critical to the maintenance and evolution of software systems. By integrating static and dynamic analysis with complementary techniques such as model checking and symbolic execution, the process provides an understanding of a system's architecture. Such an understanding is critical for making informed decisions about system maintenance and refactoring efforts, thereby supporting the long-term sustainability of the software [CDC11].

2.6.2. Model-Driven Reverse Engineering

MDRE uses the principles of model-driven software development to facilitate the analysis and understanding of existing software systems. This is achieved by abstracting the structural, behavioral, and other attributes into models [RS04]. Unlike traditional reverse engineering techniques, which focus on minute code-level subtleties, MDRE prioritizes the construction of models that encapsulate important system attributes. This approach provides a scalable and reusable framework for reverse engineering. The process typically begins by transforming a software system into a set of models representing various software development artifacts. This involves parsing the code base to extract structural and behavioral information, which is then represented using modeling languages such as UML [UML], PCM [Reu+16] or Knowledge Discovery Metamodel (KDM) [KDM].

The KDM is an OMG standard designed to provide a framework for representing knowledge extracted from existing software systems [KDM]. The KDM offers a representation of software artifacts, their relationships, and their behaviors, facilitating the process of reverse engineering. This capability is critical for organizations seeking to analyze, understand, and transform software systems [PGP11]. Several tools and platforms are available to support KDM, including MoDisco, an Eclipse-based model-driven reverse engineering toolkit [Bru+10]. MoDisco is a tool that facilitates the extraction of KDM models from existing systems, enabling reverse engineering activities that generate high-level models for understanding complex systems [Bru+14].

An aspect of MDRE is meta-modeling, where metamodels describe the structure and constraints of the models. This supports consistency and enables automation throughout the reverse engineering process. Tools' role in automating model extraction, transformation, and validation tasks is critical for managing large and complex systems. To validate and analyze models, it is necessary to compare them to their respective metamodels to confirm adherence to the expected structure and semantics. This process supports model integrity and enables accurate system analysis [RAZ17].

MDRE enables system evolution by facilitating the understanding, documentation, and communication of complex system designs through the high-level abstractions provided by the models. These models provide a foundation for various reengineering tasks, including migrating systems to new platforms, refactoring code, and integrating new functionality. The refined models can sometimes be turned into code, facilitating a partial or complete system rebuild. By emphasizing the principles of abstraction and formalization, MDRE enables engineers to prioritize the system's basic components, promoting informed decision-making and increasing the overall efficiency of the reverse engineering process [Fav10].

2.6.3. Parametric Behavior Reverse Engineering

Parametric dependencies are critical for encapsulating the relationships between input data and the variables that affect behavioral models, including loop iterations, branch conditions, and data transfers to required services [KBH07; Bec+10]. These dependencies manifest themselves through method calls, argument passing, and return values and are the foundation of control and data flow modeling. Parametric dependencies are distinguished from slices by directly mapping input parameters to typed values according to the precise grammar of stochastic expressions [CKK08].

These dependencies are critical for several functions, including estimating resource requirements, defining control flows based on interactions with external components, and facilitating data management across system components. For example, control flow dependencies can use expressions derived from input data characteristics to determine the number of loop iterations. Alternatively, they can incorporate method return values to determine branch conditions [BKK09].

In the PCM, parameter characterization reduces data complexity and simulation time by focusing on performance-relevant attributes, such as byte size, rather than complex values. This level of abstraction is valuable for handling large data sets and understanding model behavior without data manipulation. Behavioral reverse engineering requires this level of abstraction [KKR10].

The SOftware MOdel eXtractor (SoMoX) SEFF Reconstruction process from Krogmann [Kro12] provides an example of applying parametric dependencies in reverse engineering software behavior to SEFFs. This approach begins by examining the methods used and identifying their parametric dependencies as components are delineated. Control flow elements not architecturally relevant are excluded from the resulting SEFF. This is accomplished using specialized tools, including parsed source code, the static architecture model, and a code decorator model. This decorator model is generated during the architectural reverse engineering phases and establishes a link between source code elements and architectural components. This is achieved by associating operation signatures with Java methods and corresponding data types with classes. This supports that changes to the source code method bodies are incrementally reflected in the SEFFs [Kro12; Reu+16].

The reconstruction of the SEFFs is performed in two phases. In the first phase, all method calls within a given technique are classified according to their targets and origins. This is done using entries within the code decorator model, supporting accurate architectural component mapping. The parent statements of these method calls are then identified as having architectural relevance, and additional method calls are recursively classified to create an interaction map [Kro12; Reu+16].

Constructing a SEFF requires identifying and mapping parametric dependencies and control flow elements, including external call actions, branches, and loops, critical to interactions outside the component. This process provides a higher level of abstraction from the source code and delineates internal actions, specifically encapsulating those method calls that do not affect external component behavior. When a process is internal to the component and involved in external interactions, reconstruction may result in the generation of explicit representations within multiple SEFFs. Alternatively, it may exhibit internally resource-intensive behavior, thereby increasing the accuracy and efficiency of the SEFF reconstruction process [KKR10; Kro12; Reu+16].

2.6.4. Static Application Security Testing

Static Application Security Testing (SAST) is a software development methodology for identifying security vulnerabilities in software code during the implementation phase without running the program. Referred to as white-box testing, these practices focus on analyzing source code to identify potential security flaws, code quality issues, and non-compliance with secure coding standards early in the development process before the code is compiled [PM04].

SAST tools use various techniques to examine code, starting with creating Abstract Syntax Trees (ASTs) or similar structures, followed by pattern matching and rule-based analysis. This enables the identification of common vulnerabilities, including buffer overflows, Structured Query Language (SQL) injection, and cross-site scripting. Integration of these tools with Integrated Development Environments (IDEs) and CI/CD pipelines is becoming common, providing real-time feedback to developers. This immediate analysis can help prevent introducing security issues early, reducing the risk of subsequent security threats [PM04].

Despite their benefits, including scalability and the ability to automate vulnerability detection, these tools face challenges. These tools are prone to generating false positives, inundating developers with alerts for non-existent vulnerabilities, and false negatives, in which real vulnerabilities go undetected. These limitations are particularly apparent in complex scenarios, such as tracing data flow through external systems or analyzing code that cannot be compiled. In addition, the need to perform resource-intensive scans on large code bases can slow development. In addition, configuring these tools to meet specific project requirements sometimes requires customization [PM04; Li+23; Sus+23].

2.7. Terminology

To facilitate the reverse engineering of software architectures, we introduce a terminology that serves as a conceptual framework for our approach and provides an understanding of its elements. The terminology is based on the functional concepts introduced earlier in this chapter. This section contextualizes the previously introduced terms and elaborates on their meaning. It also introduces new terms for understanding the reverse engineering approach. The terminology is intended to provide a common language for discussing the concepts and techniques involved in reverse engineering software architectures. It is also designed to serve as a reference point for future research.

The *software architecture reverse engineering* process can be conceptualized as a dual effort, involving both recovery and reconstruction of the original architecture. The *software architecture recovery* process consists of extracting the architectural structure of a software system from its existing code base, configuration, and other operational artifacts. The goal is to identify the components (such as modules, classes, or packages) and their interactions to represent the high-level structure of the system. The *software architecture reconstruction* is a process rather than simply recovering it. The extracted data is refined and transformed into an architectural model that conforms to contemporary architectural standards or patterns. Thus, reverse engineering derives the software architecture from the existing product without prior knowledge. This term implies disassembling a system to understand and identify its components and their relationships. This information can then be used to rebuild the system or verify that it conforms to certain design principles.

The conceptual approach presented in this thesis is based on standards, including the "*OMG Unified Modeling Language (OMG UML)*" [UML] and the "*ISO/IEC/IEEE 42010:2022(E) – Systems and software engineering – Architecture description*" [I42010]. Among other concepts, the basic terminology of model elements or artifacts has been adopted from the UML. These terms facilitate a structured approach to discussing software architectures by defining a hierarchy from general meta-level structures to specific instances. Similarly, we adopt the view types defined in the International Organization for Standardization (ISO) and other sources. These terms describe components and interactions within software systems and support consistency with internationally recognized standards. For an overview of terminology, concepts, and their interrelationships, see Figure 2.6.

Our approach aligns with existing modeling standards by integrating and extending these terms. It presents a technique for transforming raw data from software artifacts into structured, meaningful architectural views. These transformations, governed by the defined rules, facilitate a progression from initial data extraction to final model representation. This supports that the resulting software architecture models are accurate, relevant, and valuable. To achieve this goal, we extend the existing terminology and introduce additional terms: *trace link*, *extraction rule*, *refinement rule*, and *finalization rule*. These newly introduced terms are central to explaining our techniques for model-to-model transformations. Each of these terms serves a specific function within the transformation pipeline.

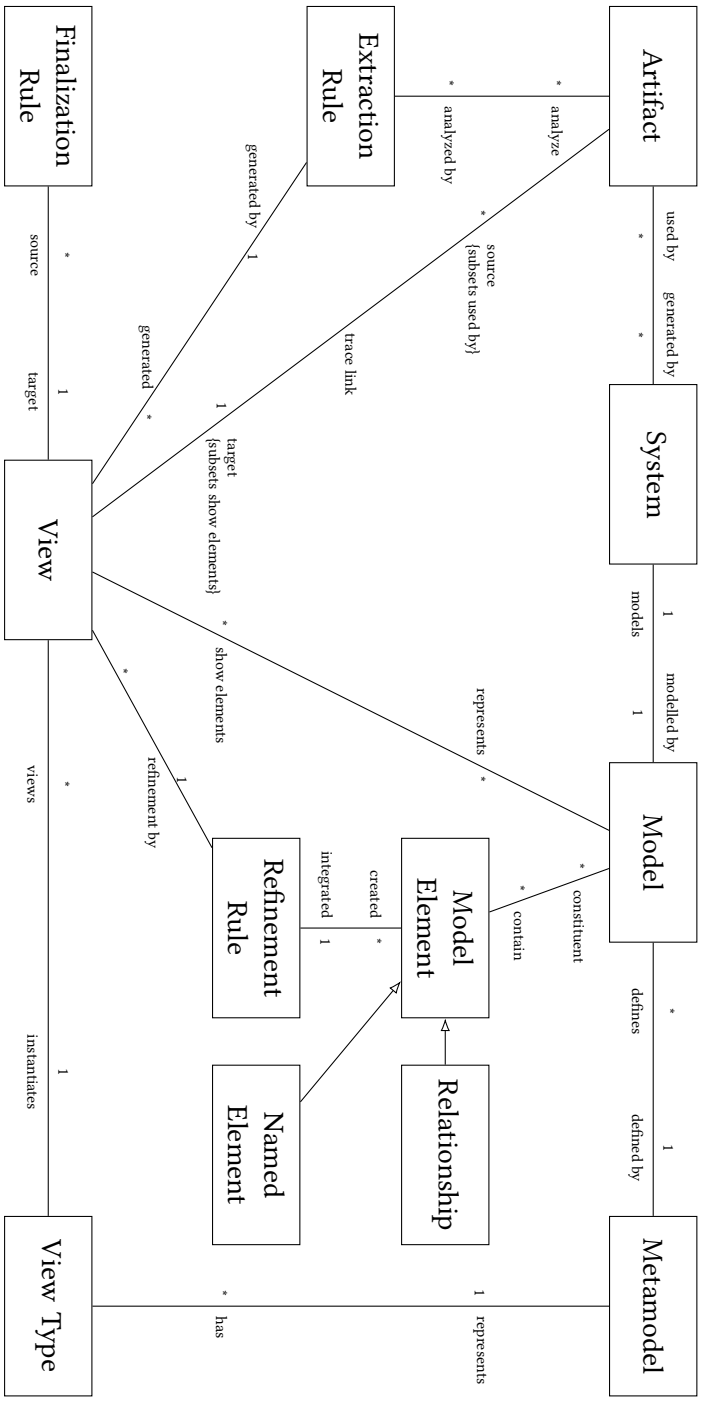


Figure 2.6.: The terminology employed in our proposed view-based reverse engineering approach.

The extraction rule represents a specific protocol for how a given concept within an artifact affects the system. It guides the extraction of relevant data from artifacts and facilitates the generation of instances of views. This rule is implemented through a model-to-model transformation, which supports the view generated as representing the underlying system parts manifested in the artifacts. A refinement rule represents an additional model-to-model transformation where the model elements present in a given view are evaluated

for compliance with predefined criteria. If the conditions are met, the view is retained; otherwise, the rule directs the generation of new model elements to be integrated into the view. This process is critical for adapting views to evolving requirements and standards. The finalization rule oversees the transformation of model elements from one view to a subsequent view. This rule filters and selects elements from the original view and merges them into a new, targeted view based on predefined conditions. This final transformation is essential in shaping the final representation of the model and ensuring that it is coherent.

The refinement and finalization rules presented here are designed for component-based software systems, although they are not limited to one system. In contrast, the extraction rules are either specific to a particular technology or to the individual system to which they have been adapted. This allows the reuse of refinement and finalization rules within the domain and extraction rules within the technology.

A trace link is conceptualized as an association that connects represented model elements regarding artifacts used or produced during the software development lifecycle or the deployment and operation of a system. This association is critical to maintaining traceability and supporting changes to development artifacts accurately reflected in the corresponding model elements [Got+12].

Part II.

View-Based Reverse Engineering

3. Reverse Engineering Approach

Our research introduces Retriever, an approach designed to address the complexities inherent in the reverse engineering of software architecture models from diverse technology-based software artifacts. The Retriever approach is underpinned by a framework capable of handling various formats and programming languages and translating them into a unified architectural model representation that facilitates system analysis and modeling. The Retriever approach is structured around two core processes: the transformation of reverse-engineered models into representations that closely mirror the original component-based architecture of the software system and the integration of model-driven reverse-engineering processes that combine and refine multiple views while resolving overlaps and redundancies between them. This integration is essential, enabling information processing from heterogeneous artifacts and supporting architectural models that are complete and coherent.

Our contributions include the development of novel methods for refining and composing extracted views that improve the applicability and accuracy of reverse-engineered software architecture models. These approaches are specifically tailored to enhance the applicability and accuracy of reverse-engineered software architecture models. By applying these approaches, our approach not only supports the extraction and refinement of structural and behavioral views from existing software artifacts, but also connects these views through a model-driven composition framework. This framework is designed to weave together various elements extracted from the artifacts, creating a cohesive and coherent architectural model.

Our approach introduces a novel refinement process that uses rule-based procedures to support the completeness and coherence of views. It involves rule-based procedures that support the completeness and coherency of the views. This refinement process is followed by a transformative step, which is instrumental in producing final output views customized to meet specific user requirements. These output views are potent tools for model-based analysis and quality prediction, providing an understanding of the software system's architecture. This approach is conceptualized to allow for exploring each step within our framework. The technical implementation, detailed in subsequent chapters, is supported by a series of subsections that analyze the terminology used within our approach. The strategies employed for extracting views from artifacts and the techniques for composing and refining these views into a final model play a role in the overall applicability and accuracy of the Retriever approach. Each of these components ensures that the approach meets the capabilities of existing approaches.

The Retriever approach is based on two concepts that collectively address the nature of software architecture. The initial concept is to develop a knowledge representation model designed explicitly for software architecture views. The model is designed to encapsulate various technologies, formats, and programming languages prevalent across diverse software artifacts. In modern software development environments, artifacts like Java or JavaScript source code, Comma-Separated Values (CSV), YAML, or JSON configuration files, and build automation tools like Maven or Gradle are typical. Our model employs model-to-model transformations, reconstructing architectural views by mapping and interpreting technology-specific relationships and conceptual frameworks embedded within these diverse artifacts. The objective is to create a representation that can adaptively cover various architectural views and their respective technological and structural views. The second foundational idea of our Retriever approach is building a framework that integrates these diverse views into a cohesive architectural model. This integration is achieved through model transformation and composition approaches designed to seamlessly map and align concepts from view-specific models to a unified architectural framework. A view composition algorithm has been implemented, which is essential in this process. This algorithm combines and refines the different architectural views gleaned from individual views and supports the idea that the resulting unified model adheres to a metamodel, enhancing both coherence and fidelity.

Our approach incorporates a series of transformations that progressively refine and integrate extracted views to operationalize these concepts. The process begins with identifying and extracting elements and relationships from the initial artifacts, then their transformation into intermediate representations conducive to analysis and integration. Subsequently, these intermediate forms are refined and merged into a unified architectural model that encapsulates the spectrum of the system's architecture. This approach is particularly productive in addressing the complexities associated with modern software systems, which involve a heterogeneous mix of technologies and architectural styles.

Our research introduces the Retriever as a novel approach to improve the comprehension and management of complex software systems through automated reverse engineering. The Retriever approach provides a mechanism for seamlessly integrating diverse elements into a unified architectural model, facilitating understanding of software systems. The Retriever approach employs the strengths of model-driven engineering and rule-based processes to provide a solution for reverse engineering software architecture models. This solution is particularly well-suited to the demands of modern software development environments, where diversity in technology and rapid changes in system design necessitate adaptable modeling approaches. The approach offers an approach for modeling and managing the complexity of contemporary software systems across various domains and technologies. The approach facilitates the continuous evolution of software architectures by enabling adjustments to the model in response to new information or changes in system requirements. Through this approach, stakeholders are equipped with the tools necessary for precise architectural analysis and quality management, facilitating better decision-making and planning in the maintenance and evolution of software systems.

The subsequent sections are based on the following authored publications: [SK19; Kir21; Kir+23a; Kir+23b; Gst+24; Kir+24a; Kir+24b]

3.1. Hourglass Paradigm

Our Retriever approach to automated reverse engineering of software architecture models is characterized by its adaptability and extensibility, which are necessary for accommodating a wide range of technologies and output models suitable for model-driven quality prediction. These attributes contribute to our framework, allowing it to process different types of inputs and generate different models without being constrained by specific metamodel dependencies. This is in a domain where the types of data input and the desired forms of analytical output can vary widely from project to project.

The core strength of our approach lies in its metamodel-independent and reusable functionality for handling model-driven information. This independence from specific metamodels improves the reusability of our strategies, making them applicable to a wide range of software engineering tasks without requiring modifications for different project requirements. This feature gives our framework a distinctive hourglass characteristic that defines its structure in terms of diversity and generality.

On one side of the hourglass, our Retriever approaches' diversity with its ability to process a wide range of inputs and generate diverse outputs. This diversity directly supports the nature of model-driven reverse engineering and model-driven quality prediction, allowing our framework to adapt to the requirements of different software systems and their architectural complexities. On the other hand, our approach is generic in its mechanisms for collecting and refining information. This generality allows the application of our strategies across projects and technologies, thereby increasing the applicability and accuracy of the framework.

Figure 3.1 illustrates our hourglass paradigm, which is organized into three processes of our approach that facilitate the extraction, refinement, and completion of software architecture information, finally leading to quality analysis. The left side of the hourglass is labeled with the name Retriever of our approach to retrieving diverse and relevant information from the software system.

The software systems are at the top of the hourglass, which is the starting point of the process. The software system is subjected to various operations to extract meaningful architectural views. At the bottom of the hourglass is the quality analysis, which represents the ultimate goal of our approach: to perform a quality analysis of the software architecture models derived from the extracted and refined views. The horizontal axis labeled *Diversity* emphasizes the importance of considering a wide range of architectural views throughout the process. The approach accurately represents the software architecture by capturing different views.

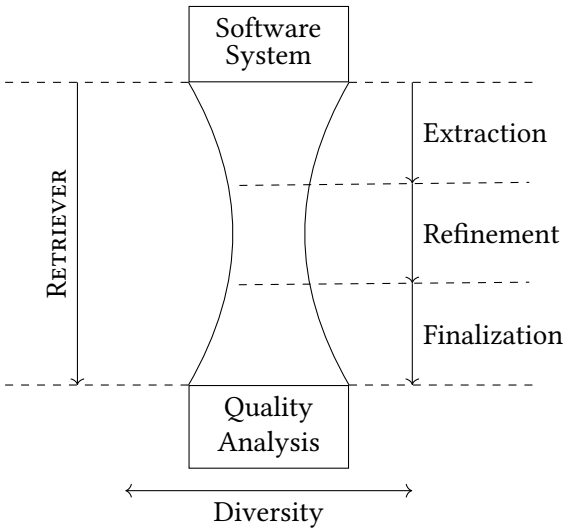


Figure 3.1.: The Retriever approach is adaptable to various technological artifacts and produces diverse analytical models, supporting information processing, project-specific adaptability, and a core set of reusable, coherent processes. This figure is adapted from the following authored publication: [Gst+24]

As information moves through the hourglass, it passes through three steps: *Extraction*, *Refinement*, and *Finalization*. These steps are shown on the right side of the hourglass.

Extraction This initial step involves identifying and extracting relevant architectural views from the software system. The goal is to abstract raw artifacts to represent different architectural views.

Refinement After extraction, the collected views are refined. This process involves filtering, organizing, and structuring the views to create coherent and meaningful architectural views. Refinement ensures that the extracted information is accurate and usable.

Finalization This step finalizes the refined architectural views. This includes validating the views, supporting their completeness, and preparing them for analysis. Finalization aims to produce well-defined architectural models that accurately represent the software system.

In summary, our Retriever approach integrates the need to handle diverse technology environments with the efficiency required for complete software architecture analysis and quality prediction. This blend of adaptability, extensibility, and reusability makes it a tool in the modern software engineering arsenal, particularly in automated reverse engineering and model-driven quality analysis. With this approach, we provide a framework that supports the exploration and refinement of software architectures and improves model-driven engineering practices across domains. The adaptability provided by different implementation options allows our approach to tailor its processing techniques to various types of input information and desired output models. Conversely, the reduced variety of implementations within the central part of the hourglass highlights our commitment

to creating a metamodel-independent core that facilitates the reuse of processes and techniques.

The following section builds on the hourglass paradigm to present an account of the distinctive processes inherent in our approach.

3.2. Process Overview

Our Retriever approach introduces refined terminology that forms the backbone of a conceptual framework designed to improve understanding and operationalization of key elements within this field. This framework uses a blend of terminology from the Unified Modeling Language (UML) [UML] and the International Organization for Standardization (ISO) [I42010], adapting and extending these terms to suit the specific requirements of reverse engineering and model-driven architecture analysis. To establish coherent terminology, we have related these terms to each other and extended the existing terminology to include trace link, extraction rule, refinement rule, and finalization rule.

The first addition to this terminology in our approach is the concept of a *trace link* [Got+12]. A trace link is an association that connects model elements depicted within a view (target) to artifacts that are either used in or result from software development processes, including deployment and operational activities (source). This linkage supports the idea that each architectural component can be accurately traced back to its origins, facilitating accountability and traceability in complex systems.

We introduce the concept of an *extraction rule*, which outlines how specific concepts are implemented within artifacts and their resultant impacts on the system architecture. These rules are executed as model-to-model transformations, creating view instances that reflect the structural and functional aspects of the system captured within the artifacts.

Another framework component is the *refinement rule*. This operates through a model-to-model transformation mechanism that evaluates whether a model element within a view conforms to set criteria. If the model element fails to meet these criteria, the rule triggers the creation of new model elements, termed *implications*. These implications are necessary adjustments integrated into the view to resolve discrepancies and support the model's integrity and compliance with the defined criteria.

Lastly, the *finalization rule* defines the techniques for transforming the elements displayed in one view into another refined view. This rule functions like a filter, selecting and assimilating model elements from the initial view into a newly formulated view based on specified conditions. This transformation is essential during the final steps of the view-based approach, as it serves to streamline and condense the model elements into a coherent and targeted representation suitable for further analysis or deployment.

Separation of concerns is important in managing the inherent complexity of developing large-scale software systems. When applied to model-based reverse engineering, this

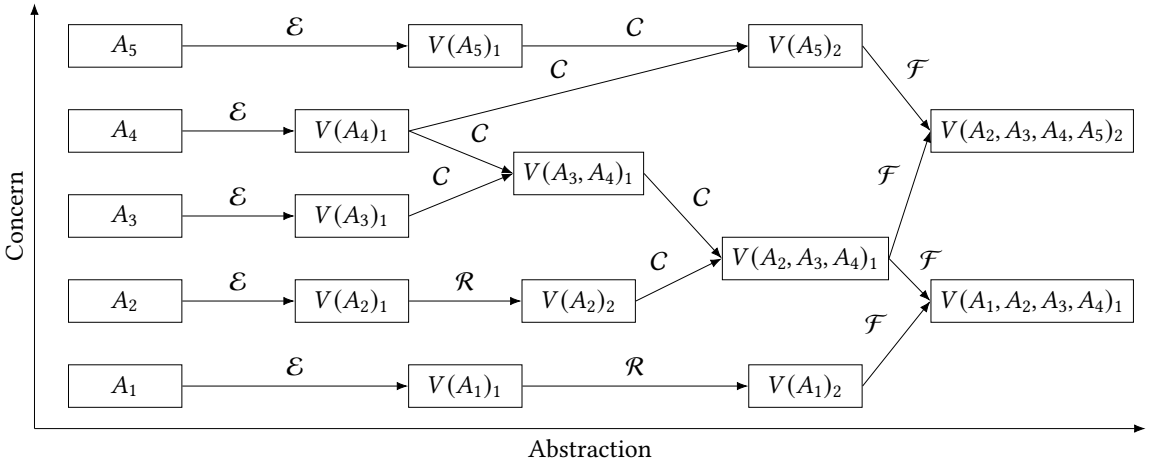


Figure 3.2.: Two-dimensional arrangement of views, where the vertical dimension represents the concerns and the horizontal dimension indicates the views' abstraction level. This figure is adapted from the following authored publication: [Kir+24a; Kir+24b]

principle facilitates the creation of multiple system models from varied views, each addressing different aspects of the system under study. Such an approach is inherently a process of refinement steps, where abstract models are progressively detailed into concrete representations that closely mirror the actual system.

An aspect of this refinement process is maintaining semantic integrity across transformations. This implies that each transformation from one model to a detailed one must retain the underlying meaning and accuracy of the information represented. Independent refinement along distinct pathways allows for targeted model improvements, addressing specific system characteristics or requirements. Nevertheless, to achieve a view of the system, these independently refined models must eventually converge through a process known as model composition.

The model composition combines at least two input models into a unified composite model that encapsulates the features of both inputs while preserving their semantic content. This process ensures that the composite model accurately represents the system and the individual input models. This process of refinement and composition is informed by transformation scenarios commonly utilized in software and data engineering, which have been adapted here for model-based reverse engineering as described by Kurtev [Kur08].

In Figure 3.2, a two-dimensional coordinate system is employed to illustrate the arrangement and progression of these views. Vertically, views are organized according to the specific problems they address, referred to as concerns. The refinement and composition of views representing different concerns represent an aspect of this process. This approach permits the coherent integration of diverse architectural views, facilitating the integration of various concerns. Each step in our model-based reverse engineering process allows views to engage with multiple concerns simultaneously, reflecting the nature of software systems. Horizontally, the views are arranged by their level of abstraction. The initial

Algorithm 3.1 Our approach's main reverse engineering process is an entry point for subsequent analysis. The algorithm is presented in three steps, which may initially appear relatively straightforward. Nevertheless, this is an intentional design choice, as the process has been divided into three steps using the divide-and-rule approach.

```
Input: E                                          ▷ Set of all extraction rules
Input: R                                          ▷ Set of all refinement rules
Input: F                                          ▷ Set of all finalization rules
Output: M_f                                       ▷ Final output models ready for consumption
  function ReverseEngineering(E, R, F)
    A ← FindAllArtifacts                          ▷ Find all artifacts from the system
    V_e ← ExtractViews(A, E)                      ▷ Extract structural with behavioral views
    M_f ← CompositionRefinement(V_e, R, F)        ▷ Composition, refinement with finalization
    return M_f
  end function
```

artifacts, designated as A, are situated on the spectrum's left. These are transformed into concrete views, defined as $V\,(A)$, through abstraction.

The transformation process begins with an extraction $\mathcal{E}$, which creates an instance of views based on the artifacts representing parts of the system. This is followed by a refinement $\mathcal{R}$, where the information extracted from the artifacts is further concretized. These refined views may then be combined with information from other artifacts or additional views. Finally, a final transformation is employed, where elements from views are selectively merged to form a new target view based on specific criteria.

In our approach, the extraction is tailored to be technology- or project-specific, reflecting the system's characteristics being reverse-engineered. Conversely, the steps of combination C, refinement $\mathcal{R}$, and final transformation $\mathcal{F}$ are designed to be generic and applicable across different component-based systems.

Our approach commences with identifying and analyzing artifacts from the software system. The artifacts mentioned above are then processed using specialized finders, which may include third-party parsers, to transform the raw data into preliminary views suitable for further transformations. This is accomplished through the FindAllArtifacts helper function, as illustrated in Algorithm 3.1. Subsequently, the views are subjected to extraction rules in the ExtractViews function. Subsequently, each extracted view is subjected to examination against predefined criteria. This refinement process ensures the architectural views are accurate and reflect the system's operational requirements. The CompositionRefinement function culminates, integrating these refined views into several final models. The algorithm's final models, designated as M_f, represent a unified representation of the individual views prepared for further analysis and consumption.

These terminological improvements and the introduction of rules for extraction, refinement, and finalization clarify the process and enrich the foundation of our Retriever approach. We have created a framework supporting modern software systems' complex nature by categorizing and processing elements through these steps. This framework

permits reverse engineering and continuous refinement of software architecture models. The generic framework facilitates the extraction step's combination of extracted views, requiring adaptations or extensions. This structured approach exemplifies our dedication to facilitating model-driven analysis and quality prediction while accommodating the diverse technological landscapes of modern software systems.

3.3. Retriever Illustration

This section presents the steps behind our Retriever approach, demonstrating its application within a Java-based project that utilizes the Spring Framework and Docker. This example has been simplified to emphasize the role of annotations in understanding and reconstructing the architecture of software systems. We support clarity and focus by reducing the instance to its components. However, this entails the omission of specific technical dependencies and details for the sake of brevity. By focusing on this streamlined example, our objective is to illustrate the application of our approach in a real-world scenario. This demonstration will demonstrate the efficacy of model-driven reverse engineering in capturing the architecture of software systems developed with contemporary technologies.

The Spring Framework is employed here as a platform for developing web applications, providing a structured environment for the `AccountController` class and `AccountServiceClient` interface. In contrast, Docker supports the idea that applications are neatly isolated in containers, facilitating a manageable deployment process. This combination of technologies represents a modern approach to software development in which multiple technologies converge to form complex systems.

The initial step of the approach preparation process involves identifying and retrieving artifacts, encompassing Java source files, Spring configuration files, and Docker deployment scripts. Each type of artifact plays a role in providing an understanding of the software's architecture.

Java Source Files These files contain the actual code determining the application's behavior. They describe how components interact within the network and the pathways through which data and control flow between different application parts.

Spring Configuration Files These files are important in defining the operational specifics of the application, including Uniform Resource Locator (URL) mappings and service configurations. They facilitate an understanding of the structure and intended functionality of the application.

Docker Files These files provide the deployment configuration necessary to comprehend the operational environment in which the application is deployed. They are important for assessing containerization and the management of the application in production environments.

Our reverse engineering approach employs these artifacts to extract and connect the system's components, exemplifying how annotations within the Spring framework can delineate component boundaries and interactions.

3.3.1. Extraction of Components with Deployments

Figure 3.3 provides an illustration of extracting the `AccountController`, a Representational State Transfer (REST) controller component, from a Java-based project employing the Spring Framework. Its `@RestController` annotation accurately identifies the controller and further delineates it through methods tagged with `@GetMapping`. This step exemplifies the approach's capacity to identify and delineate the functionalities of components within the software system.

In our simplified example, while the Java class `Account` is utilized as a data transfer object and does not correspond directly to a system component, it nevertheless plays an indispensable role. It delineates the parameters for the service endpoint, thereby demonstrating how seemingly ancillary classes contribute to the system's operational dynamics. Similarly, the extraction process incorporates insights from the `account-service.yml` configuration file, which specifies the context path for the `AccountController`. This path influences the base URL for related endpoints, impacting both the deployment and accessibility of the service.

The extraction rules have been designed to adjust the URL paths by appending the context path detailed in the configuration file to the paths outlined in the `@GetMapping` annotations. This adjustment supports the idea that the endpoint URLs represented in the view are congruent with those in the production environment. Such precision in extracting and modifying URL paths improves the resulting software architecture model. For example, in Figure 3.3, the mapping path `/{name}` is adjusted to `/accounts/{name}` to reflect the context path specified in the configuration file.

By integrating these configurations, our approach captures the application's structural and behavioral views and deepens our understanding of its deployment context. This extraction and integration process ensures that the reconstructed architectural views are dependable and reflect operational reality. How artifacts are extracted and analyzed illustrates the precise artifact handling involved in creating accurate and functional software architecture views. This consideration and handling of each component and configuration within the system exemplifies the efficacy of our approach in facilitating an insightful reverse engineering process.

Figure 3.4 illustrates the identification and extraction of the `AccountServiceClient`, a component that uses the `@FeignClient` annotation. This annotation, part of the Spring Cloud library [VMw24], provides a declarative method for constructing web service clients, simplifying the development and integration of REST service calls by abstracting the complexities of Hypertext Transfer Protocol (HTTP) request construction and response handling. The accompanying code snippet illustrates that the `AccountServiceClient` class employs the `@GetMapping` method to define a specific service call. This method annotation

```java
@RestController
public class AccountController {
   @GetMapping(path = "/{name}")
   public Account getAccount(String name) { }
}
```

AccountController.java

```yaml
server:
   servlet:
      context-path: /accounts
```

account-service.yml

```
if (cl.isAnnotated("RestController")) {
   Component co = createComponent(cl.name);
   for (Method m : cl.annotatedWith("GetMapping")) {
      String path = config.contextPath + m.getValue("GetMapping");
      co.addEndpoint(fullPath, m.name);
   }
}
```

Example of an extraction rule for a REST controller.

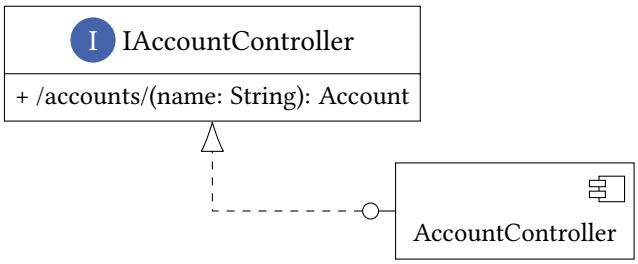


Figure 3.3.: The `AccountController`, annotated as a REST controller with `@RestController` and `@GetMapping`, is identified as a component. The context path from the `account-service.yml` is prepended to the endpoint paths, indicating the base URL for related requests. This figure is adapted from the following authored publication: [Kir+24a; Kir+24b]

maps HTTP requests to specific handler methods, thus facilitating the direct retrieval of information based on the specified URI. In this context, the `AccountServiceClient` interacts with the account service through the straightforward interface method `getAccount`, which retrieves account details based on a user's name. This is important for our approach, as it is identified as a service client component by the extraction rules designed to parse the annotations and method definitions.

The abovementioned rules identify this class as a service client component due to its `@FeignClient` annotation. These rules have been designed to parse the annotations and method definitions, generating a corresponding component view in the software architecture. As illustrated in the accompanying pseudocode, the rule iteratively checks for classes marked with `@FeignClient`, and for each such class, it further inspects methods

```java
@FeignClient(name = "account-service")
public class AccountServiceClient {
   @GetMapping(value = "/accounts/{name}")
   String getAccount(String name);
}
```

AccountServiceClient.java

```
if (cl.isAnnotatedWith("FeignClient")) {
   Component co = createComponent(cl.name);
   for (Method m : cl.annotatedWith("GetMapping")) {
      co.addServiceCall(m.getValue("GetMapping"), m.name);
   }
}
```

Example of an extraction rule for a REST client.

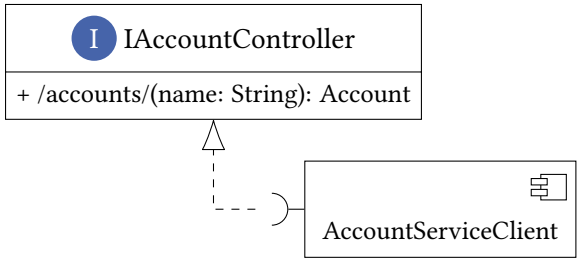


Figure 3.4.: The `AccountServiceClient` is identified as a service client component, using the `@FeignClient` annotation from the Spring Cloud library, simplifying REST service calls by abstracting HTTP request and response details. This figure is adapted from the following authored publication: [Kir+24a; Kir+24b]

annotated with `@GetMapping`. A service call is registered within the component view for each matching method, indicating an operational dependency on an external service.

Figure 3.5 depicts the Docker-based deployment architecture for the two example components: the `AccountController` and the `AccountServiceClient`. The Docker configurations specified in the Docker files determine the runtime environments for each component. The initial Docker file delineates the environment for the `AccountController`, which is a REST service provider. This file establishes a Java runtime environment and deploys the service's executable jar file to the container. Similarly, the second Docker file configures the environment for the `AccountServiceClient`, supporting it to operate as a client consuming the REST Application Programming Interface (API) provided by the `AccountController`.

The extraction rules embedded in our approach analyze the Docker files to extract deployment-related information. These rules map each component – the `AccountController` and the `AccountServiceClient` – to its respective deployment node. This mapping is essential, as it illustrates the physical distribution of components across different nodes in the system's architecture. This approach aligns with the principles of containerization and microservice architectures, where components' isolation and independent deployment are essential.

```
FROM java:8-jre
ADD ./target/account-controller.jar /app/
CMD ["java", "-jar", "/app/account-controller.jar"]
```

Example Dockerfile for AccountController

```
FROM java:8-jre
ADD ./target/account-service-client.jar /app/
CMD ["java", "-jar", "account-service-client.jar"]
```

Example Dockerfile for AccountServiceClient

```
for (Dockerfile d : allDockerfiles()) {
   Deployment node = createDeployment(d.parentFolder, d.name);
   node.addComponents(d.app.name);
}
```

Example of an extraction rule for Dockerfiles.

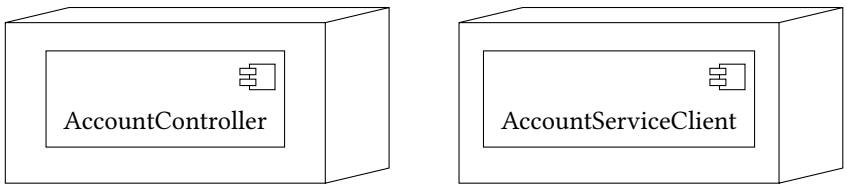


Figure 3.5.: In the containerized setup, the `AccountController` and `AccountServiceClient` are deployed on separate nodes, with Docker files defining each component's environment. The extraction rule assigns each component to a deployment node based on Docker configurations. This figure is adapted from the following authored publication: [Kir+24a; Kir+24b]

The automated extraction of this deployment information is facilitated by the rule shown in the pseudocode, which iterates over Docker files to create deployment nodes for each identified component. This rule encapsulates the logic for assigning components to specific nodes based on the settings defined in their respective Docker files. This supports the idea that the system architecture accurately represents each component's deployment context. Consequently, the extraction and mapping processes are essential for accurately modeling the deployment architecture in a containerized environment. By capturing and integrating this information, our approach enables an understanding of the system's deployment strategies and operational dynamics for architecture analysis and optimization.

3.3.2. Refinement and Composition

Figure 3.6 illustrates the application of a series of model transformations in our automated reverse engineering approach. The series of view transformations comprises one composition rule and two subsequent refinement rules. These rules aim to integrate and improve the structural integrity of the architectural views derived from various software artifacts. This sequence of composition and refinement rules demonstrated through code examples illustrates our approach's capacity to generate accurate architectural views. We have

assumed that two components are connected through appropriate interfaces for simplicity. However, since this is not always the case, our approach could also evaluate the Docker file to determine if two compute nodes are connected. The views thus generated reflect the software architecture and facilitate ongoing analysis and optimization tasks within a model-driven engineering framework.

The steps commence with a composition rule amalgamating Docker deployment data with architectural elements extracted from Java files. This integration is essential, as it aligns the deployment environment described by Docker with the logical structure defined in the Java source code, specifically Spring components. The composition rule facilitates the programmatic linking of Docker containers to corresponding Java classes or components with the same qualified name. This rule establishes a preliminary architectural view in which each component is correctly situated within its operational context.

Once the initial composition has been established, the first refinement rule is applied to ensure that components intended to interact are appropriately connected. This step entails the establishment of connections between the provided and required interfaces of components that share common endpoints, as indicated by matching URLs. The rule verifies the required interfaces in each component and identifies correspondingly provided interfaces within the deployment context, thereby establishing direct links. These connections are essential for accurately simulating real-world interactions within the software system.

Subsequently, a second refinement rule is implemented to connect the deployment nodes that house the interacting components. Given that the `AccountServiceClient` is connected to the `AccountController` through matched interfaces, this rule identifies and links their respective deployment nodes. This step is essential for accurately reflecting the physical distribution of the system's components across different nodes in the deployment architecture.

3.3.3. Final Architectural Representation

Figure 3.7 depicts the final step of the process, which culminates in transforming refined and assembled architectural views into a final output model. This view is specifically designed to meet user-defined requirements, such as UML diagrams for documentation. Integrating components and deployment information into a UML diagram via the example finalization rules improves the visual comprehensibility of the architecture and supports that aspects of the system's architecture are conveyed. For example, using UML as a format for output could facilitate understanding and communication among stakeholders, thereby enhancing collaborative development and architectural documentation processes.

This finalization step illustrates converting architectural relationships and elements, such as the `AccountController` and `AccountServiceClient`, into their corresponding UML representations. This step reveals the structural connections between these elements and the functional connections between them. The finalization step employs a rule to depict each component within the UML metamodel. This rule iterates over each component extracted and refined from the software system's architecture, rooted in the combined information

```
for (Component c : allComponents()) {
   for (Deployment d : allDeployments()) {
      if (d.contains(c.name)) { link(d, c); }
   }
}
```

Example of a composition rule.

```
for (Component cl : allComponents()) {
   if (cl.hasRequiredInterface()) {
      Interface r = cl.requiredInterface;
      for (Component co : allComponents()) {
         if (co.hasProvidedInterface(r)) {
            linkInterfaces(co.providedInterface, r);
         }
      }
   }
}
```

Example of a refinement rule for links between components.

```
for (Deployment d : allDeployments()) {
   for (Component c : d.allComponents()) {
      if (c.hasLinkToOtherComponent()) {
         link(d, c.getLinkedDeployment());
      }
   }
}
```

Example of a refinement rule for links between deployments.

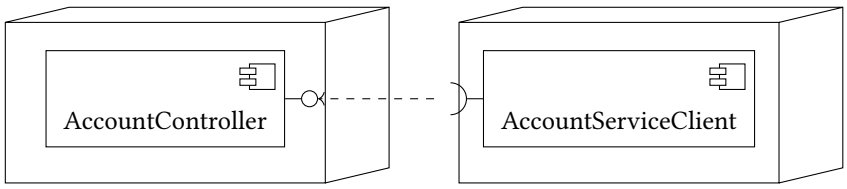


Figure 3.6.: After component extraction, Docker deployment information is merged with architectural data from Java files and then refined by connecting components. This figure is adapted from the following authored publication: [Kir+24a; Kir+24b]

from Java source files, Spring configurations, and Docker deployment files. The endpoints associated with each component are transformed into UML endpoints to represent the communication paths that were initially derived from the source code and configurations. The deployment information, which is for comprehending the operational environment of each component, is similarly transformed into UML deployment nodes. This mapping supports that the final UML diagram is a visual representation and an accurate and actionable model that reflects the system's deployment architecture, component structure, and behavior.

```
for (Component component : allComponents()) {
   UMLComponent umlComponent = createUMLComponent(component.getName());
   for (Endpoint endpoint : component.allEndpoints()) {
      UMLEndpoint umlEndpoint = createUMLEndpoint(endpoint.getPath());
      umlComponent.addUMLEndpoint(umlEndpoint);
   }
   Deployment deployment = component.getDeployment();
   UMLDeployment umlDeployment = createUMLDeployment(deployment.getName());
}
```

Figure 3.7.: The following example illustrates a finalization rule for UML that iterates over each component extracted from the software system's architecture. This process transforms the endpoints associated with each component into UML endpoints and the deployment information into UML deployment nodes. This figure is adapted from the following authored publication: [Kir+24a; Kir+24b]

4. Extracting Views from Artifacts

Automated extraction of views of existing software artifacts – such as source code and configuration files – is one of the critical tasks in unraveling and documenting the complex structure and behavior of software systems. This time-consuming task aims to transform software systems' complex architecture into coherent, analyzable views. Algorithms are used to analyze and interpret the underlying architecture of the artifacts generated during the software life cycle. Our Retriever approach, exemplified by Algorithm 4.1, embodies an approach to extracting structural and behavioral views that provides a view through which the system architecture can be examined.

Extraction for module decomposition views concerns the system's static structure. It maps code structures to architectural meanings and identifies components and interactions based on static code aspects. Conversely, usage extraction rules prioritize dynamic interactions, such as method calls and data flows. The diverse focus of these approaches exemplifies rule-based methods for capturing and reconstructing software architectures from software development artifacts.

The representation of software artifacts, such as classes, in a model depends on the specific technology used and the artifact's functionality. This differentiation is controlled by technology-specific extraction rules tailored to interpret software artifacts based on their role within the system architecture. These rules are adapted to the architectural paradigms of different technologies, ensuring that they respond accurately to technological differences. For example, in frameworks such as Spring, a class may be called a component if it is annotated to indicate management by the Spring container. However, not all classes are components; some function as data transfer objects or utility classes, highlighting the need for extraction rules to accurately reflect the system's architecture.

Rules that can identify specific architectural styles and patterns, such as the model-view-controller or microservices, do so by mapping specific programming constructs to architectural elements. The manifestation of extractable styles and patterns in code is typically explicit, occurring through annotations or configuration files for naming conventions. Architectural frameworks facilitate the discovery of these patterns, making them relatively easy to identify. In contrast, abstract patterns, which may leave no discernible code signature or depend heavily on developer intent, present challenges for extraction rule implementations.

Algorithm 4.1 begins by generating structural views, which lay the groundwork by analyzing the system architecture through different views. These views focus on the modular components, their interfaces, or ports, and the dense web of relationships – be they dependencies, connectors, or distribution relationships between compute nodes and components

Algorithm 4.1 The algorithmic step of extracting structural and behavioral views from given system artifacts.

Input: A ▷ *Set of all artifact views*
Input: E ▷ *Set of all extraction rules*
Output: B ▷ *Extracted structural with behavioral views*

```
1: function ExtractViews(A, E)
2:     L ← ExtractStructuralViews(A, E)
3:     B ← ExtractBehavioralViews(L, E)
4:     return B
5: end function
```

– that bind these elements together. This step is critical for visualizing the physical and logical configuration of the system, showing the modular organization and interconnectivity of the components, and setting the stage for subsequent investigations of the software architecture.

Once the structural views have established a foundation, the Retriever approach creates behavioral views based on the abovementioned structural elements. This step involves capturing the dynamic interactions between components, abstracting the externally visible actions of component services to reflect the internal behavior, and mapping the complex exchanges between the provided and requested component interfaces. Such abstraction not only clarifies the behavioral dynamics of the system but also reveals the complex relationships and interactions underlying the system's functionality, providing a granular view of the behavioral mechanics of the system.

This structured transition from the structural to the behavioral view, facilitated by Algorithm 4.1, requires the creation of trace links L. These trace links are critical for correlating the implementations of code classes, interfaces, and methods with their respective architectural abstractions, such as components, interfaces, and signatures. Through mapping, these trace links support the idea that each architectural element derived from the code is accurately represented in the architectural model, increasing the coherence and fidelity of the extracted views. This process culminates in merging structural and behavioral views into a representation of the software system B.

By analyzing and transforming software artifacts into structured views, our Retriever approach reverse engineers the software architecture and improves the understanding of the software system. This dual approach to extracting structural and behavioral views gives architects, developers, and stakeholders the tools to analyze, navigate, and evolve the system architecture with precision and insight. In addition, this step provides a framework for understanding and documenting the complex architectures that underpin today's software systems, streamlining the extraction of architectural views and increasing the accuracy of architectural documentation.

The two functions ExtractStructuralViews and ExtractBehavioralViews play a central role in extracting views from artifacts, and their descriptions are provided in Sections 4.1 and 4.3. We will go into their inner workings and describe how they process

inputs to generate their respective outputs. Understanding these functions will explain how we extract software artifacts' structural and behavioral views.

> The subsequent sections are based on the following authored publications: [SK19; Kir21; Kir+23a; Kir+23b; Kir+24a; Kir+24b]

4.1. Extracting Structural Views

As part of our Retriever approach, one of the subtasks is the extraction of structural views from various software artifacts, including but not limited to source code and configuration files. This process is essential for describing the architecture of software systems. It eases the transition from textual descriptions and specifications to models, enabling improved analysis and refinement using model-driven development tools. Such a transformation process is essential not only to represent the relationships and dependencies between different components but also to clarify their distribution across different computing nodes, thus raising the analysis to a higher level of abstraction.

At the heart of this process is the practice of reverse engineering, which requires an understanding of the technologies embedded in the software system. Reverse engineering enables in-depth modeling of the knowledge associated with the technologies used, revealing the mechanisms by which the components are implemented. This insight is critical to understanding the architecture influenced by the choice of specific technologies and indicates the inherent complexity of the architecture.

Articulating model transformations – realized through mapping rules – plays an essential role in this context. These rules skillfully translate the various aspects of the system design into models, capturing how different concepts are realized in other technologies and how they impact the architecture. This modeling step is critical for communicating the facets of system architecture and provides an understanding of the technological underpinnings and their architectural implications.

Our Retriever approach leverages the adaptability of extraction rules to facilitate the reverse engineering of software architecture models from structural views of artifacts. These extraction rules can consider different technology ecosystems and project requirements and support our approach to analyzing component structures, dependencies, and distribution. These rules can integrate domain-oriented modeling with explicit mapping rules and support the transition from textual artifacts to model elements through their model transformations and traceability links.

4.1.1. Apply Rules to Generate Views

Our technique for rule-based extraction of structural views from software artifacts is described in Algorithm 4.2, which serves as a blueprint for extracting views of the structure

Algorithm 4.2 Our rule-based algorithm for extracting structural views with the corresponding trace link from the given artifacts.

Input: A ▷ *Set of all artifact views*
Input: E ▷ *Set of all extraction rules*
Output: L ▷ *Extracted structural views linked to artifacts*

```
1: function ExtractStructuralViews(A, E)
2:     L ← { }
3:     for all a ∈ A, r ∈ E do
4:         if r(a) then
5:             L ← L ∪ (E_r(a), a)
6:         end if
7:     end for
8:     return L
9: end function
```

from a defined set of software artifacts. This approach is based on an evaluation of the Cartesian product $A \times E$ formed by combining all available artifacts A with a set of extraction rules E. The core of our approach is the extraction $\mathcal{E}$ that creates an instance of views based on the artifacts that represent parts of the system.

This extraction focuses on the use of a predicate $r(a)$ to each artifact $a \in A$ against each rule $r \in E$. This predicate determines the applicability of a rule to an artifact and acts as a gatekeeper to support the idea that only relevant transformations are performed. If this condition is satisfied, the specified rule is executed on the artifact, and a transformation process is started, culminating in generating a view $v \in V$. This process, symbolized by $\mathcal{E}_r(a) \rightarrow V$, is essential for extracting views that are not only representative of the system architecture but also suitable for further analysis and integration into a coherent architectural model. Each view extracted from an artifact is stored in a Set L along with a trace link. Each extracted view is associated with a trace link, which records its origin and establishes a link between the view and the artifact from which it was derived. This trace link acts as a critical identifier that precisely defines the relationship between the structural view and the original artifact, preserving the origin and context of each transformation.

The set L represents a repository aggregating all extracted views and their corresponding trace links. This aggregation is essential for several reasons. First, it provides a structured means of storing the extraction process results, allowing stakeholders to navigate and understand the transformation process. Including trace links in this set offers an understanding of how each view correlates to specific aspects of the source code or configuration files. This traceability is essential for validating the integrity of the extracted views. It supports that each structural element within the view can be directly traced to its empirical origin within the software artifacts.

At the end of the extraction process, the set L is returned as the function's output. The return of this set marks the completion of a critical step in the reverse engineering process. The contents of this set – both the extracted views and their trace links – serve as the basis for subsequent behavior extractions, in which the architectural model can be further refined

and analyzed. It also serves as a basis for decisions regarding the system's optimization or maintenance.

Having outlined the process of applying rules to generate structural views, we now focus on how these rules facilitate the identification of component structures within the software system.

4.1.2. Component Views

This process aims to identify the component structure of the software system, reveal the dependencies that connect these components, and transform their distribution across computing nodes. Such transformation is an essential part of our Retriever approach, as it facilitates the transformation of textual descriptions or specifications of a system into structured views suitable for analysis and processing by other model-driven development tools and techniques. This transformation marks the transition to higher levels of abstraction that enable a deeper understanding and analysis of the software architecture.

A critical aspect of this transformation phase is the reverse engineering process, which is supported by knowledge modeling of the technologies used in the software system. This process is instrumental in uncovering how components are realized with specific technologies, thereby revealing their architectural implications. This approach encapsulates the essence of system design into understandable views through mapping rules by applying model transformations. These mapping rules are essential to our approach because they describe the implementation of concepts within the technology framework and their impact on the overall system architecture. These rules are either technology-specific, i. e., they consider the intricacies of specific technologies, or project-specific, i. e., they allow the approach to be adapted to the specifics of individual software projects. This dual nature of the rules supports the adaptability and reusability of the approach across different technology landscapes and software projects.

4.1.3. Trace Link Views

An aspect of this extraction approach is creating and managing trace links. When created, each extracted view is associated with a trace link, which records its origin and reconnects the view to the artifact from which it was derived. These trace links, stored in the L set, serve as an archiving mechanism that preserves the origin of structural views and facilitates traceability and accountability in the architectural analysis process. The collection of these trace links and the views to which they are assigned form a repository that is returned at the end of the function. This repository encapsulates the extracted views' completeness and associated trace links, providing a granular and transparent mapping of software artifacts to structural elements.

These trace links can be used to trace how classes, interfaces, or methods in the source code are mapped to structural elements such as components, interfaces, or signatures.

These trace links act as archival references to the original artifacts, supporting the integrity and traceability of the extracted structural elements. When the extraction rules identify components or interfaces within the artifacts, a trace link is instantiated from classes to components. Similarly, detecting signatures triggers the creation of trace links from methods to signatures. Trace links are not limited to the extraction step; they are also needed in subsequent steps, such as composition, refinement, or finalization of the architectural model. Critically, our approach incorporates trace links to document each transformation from artifact to model element. These links provide an audit trail, supporting traceability and preserving the integrity of the structural view.

4.2. The Rule Concept

Reusability influences software quality in software engineering. This principle is motivated by developing more reliable and cost-effective systems. Achieving software reuse involves various strategies, i. e., through libraries and frameworks. These tools do more than provide a structured environment for application development; they inherently embody segments of the software architecture itself.

The essence of our proposed rule concept lies in adopting a rule-based approach tailored to encapsulate the domain knowledge inherent in the technologies employed during component-based software development. This effort aims to use this knowledge to facilitate the reverse engineering of software architectures from artifacts. Such domain knowledge could, for example, describe how a component is realized within a particular framework, thus capturing the properties and impact of different technologies on a system's architecture.

Our rule concept relies on a knowledge representation model for domain knowledge – understanding how specific technologies implement architectural concepts – to reconstruct the architecture of systems using those technologies. This reconstructive process inherently yields a static structural view of the system under study, derived from examining existing text-based artifacts such as source code, deployment descriptors, and other configuration files. These artifacts and predefined, system-agnostic domain knowledge about technologies form the input for our model-driven reverse engineering framework. This framework is designed to facilitate the simple and reusable encapsulation of domain knowledge, thereby enhancing the reusability of the entire process.

The key to increasing the reusability and efficiency of this approach is the explicit delineation of architectural elements, including components, interfaces, and connectors, commonly encountered in component-based software development. Traditionally, these concepts may be implicitly woven into the fabric of reconstruction mechanisms. However, by explicitly defining these concepts within our domain knowledge framework, we seek to reconstruct the system's architecture, providing insights into the interplay and implementation of individual components as dictated by the technologies employed.

The core of our rule concept is the formulation and application of technology-specific rules. These rules describe how different technologies affect the system architecture and their implementation in an architectural model. Developed through a thorough analysis of each technology's characteristics, these rules are instrumental in identifying how specific concepts are instantiated and their consequent impact on the system architecture. Manifested through model-to-model transformations, these rules provide a mechanism for identifying and mapping architectural elements according to the patterns dictated by the underlying technologies.

4.2.1. Formal Approach

Rules are formulated to encapsulate how specific concepts are implemented within a given technology or project, while highlighting the resulting impact on the system's architecture. Central to operationalizing these rules is their expression as model-to-model transformations, which are instrumental in translating the details of technological implementations into comprehensible architectural views.

The development of technology-specific rules is rooted in a thorough analytical process that examines the patterns inherent in each technology and then aligns them with the constituent elements of the architectural view. This analytical effort supports the idea that the rules encompass facets of the technology under consideration, providing a framework for identifying and extracting architectural view elements relevant to projects using the technology. These rules facilitate standardization of the architectural view extraction process and provide a blueprint for achieving an architectural view instance, assuming developers adhere to the technology specifications.

Notwithstanding the foundational nature of technology-specific rules, our Retriever approach demonstrates applicability by accommodating project-specific rules. This adaptability allows the rule set to be tailored to individual software projects' requirements, thus enabling an exploration of component implementation within specific projects. The project-specific rules augment the technology-specific rules, increasing the granularity of the derived architectural views.

The choice of transformation language is critical to articulating and implementing these transformation rules. Within our framework, Xtend [Bet16; Ecl24d] is used for this purpose due to its adaptability in structuring rules and its support for both imperative and declarative definitions. This capability of Xtend allows for a rule definition process that accommodates a wide range of rule structures and logical expressions.

A simplified representation of the framework underlying our rule-based approach is provided by a Backus-Naur Form (BNF) [Bac59; Bac+63] in Listing 4.1. The BNF formalism is a tool for describing the structure of transformation rules, focusing on the elements of those rules. This BNF delivers a conceptual framework for formulating transformation rules and details the syntax and structure required to define these rules within our approach. Despite its simplified nature, this BNF contains the core production rules essential for the formal definition of transformation rules. However, it is recognized that the BNF

representation is an abstracted subset of potential solutions, focusing only on the most salient production rules and omitting specifications of condition formulation and function parameters.

By converging the analytical depth of technology- and project-specific rules with the expressive of Xtend [Bet16; Ecl24d] within a structured BNF framework, our approach represents a distinct instance for automated reverse engineering of software architecture views. This process emphasizes the importance of understanding technology's impact on architecture but also advocates adaptability and precision in capturing the architectural essence of software systems.

```
⟨Document⟩     ⊨  ⟨Rule⟩ | ⟨Rule⟩ ⟨Rule⟩
⟨Rule⟩         ⊨  ⟨Iteration⟩ ⟨Predicate⟩ ⟨Identification⟩
⟨Iteration⟩    ⊨  for( ⟨Variable⟩ : ⟨Query⟩ ) |
                  ⟨Iteration⟩ ⟨Iteration⟩
⟨Predicate⟩    ⊨  if( ⟨Condition⟩ ) |
                  ⟨Predicate⟩ ⟨Predicate⟩
⟨Query⟩        ⊨  ⟨Variable⟩.accessView⟨Parameters⟩ |
                  ⟨Variable⟩.declarationView⟨Parameters⟩ |
                  ⟨Variable⟩.interfaceView⟨Parameters⟩ |
                  ⟨Variable⟩.invocationView⟨Parameters⟩ |
                  ⟨Variable⟩.packageView⟨Parameters⟩
⟨Condition⟩    ⊨  ⟨Variable⟩.containsField⟨Parameters⟩ |
                  ⟨Variable⟩.containsMethod⟨Parameters⟩ |
                  ⟨Variable⟩.extendsType⟨Parameters⟩ |
                  ⟨Variable⟩.hasAnnotation⟨Parameters⟩ |
                  ⟨Variable⟩.isPublic⟨Parameters⟩
⟨Identification⟩ ⊨  ⟨Variable⟩.toAggregation⟨Parameters⟩ |
                  ⟨Variable⟩.toAllocation⟨Parameters⟩ |
                  ⟨Variable⟩.toAssociation⟨Parameters⟩ |
                  ⟨Variable⟩.toComponent⟨Parameters⟩ |
                  ⟨Variable⟩.toContainer⟨Parameters⟩ |
                  ⟨Identification⟩ ⟨Identification⟩
⟨Variable⟩     ⊨  Xtend Variable identifiers
⟨Parameters⟩   ⊨  Xtend method parameters
```

Listing 4.1: The simplified BNF is the conceptual model for the knowledge representation model of our extraction rules.

Our Retriever approach to automated reverse engineering of technology-induced software architecture views is encapsulated in the `<Document>` construct, which serves as an aggregation of transformation rules defined within a simplified BNF syntax [Bac59; Bac+63]. This framework provides a structured rule concept for performing a series of operations on software artifacts to interrogate, evaluate, and finally identify architectural constructs based on iterative analyses. Each rule is designed to perform specific functions, from discovering classes and methods within the codebase to identifying architectural components and their interactions. This facilitates the decomposition of large software systems into analyzable views.

Central to this rule concept is the `<Iteration>` construct, which introduces a mechanism for iterating over elements returned by a `<Query>`. These queries are designed to target specific views or facets of the software artifacts, such as `declarationView` for type declaration definitions or `interfaceView` for interface declarations, allowing for an examination of the software system. This iteration mechanism is essential for navigating through large codebases, ensuring that every element critical to the architectural view is considered.

The `<Predicate>` construct, which is central to the functionality of the BNF, allows for the conditional evaluation of elements within each iteration. Using `<Condition>` expressions, predicates are identified based on the properties of each component under consideration. Conditions can include various checks, including but not limited to annotations, inheritance relationships, or the presence of public modifiers. This conditional logic is critical in selectively filtering elements, focusing on those with specific architectural importance for subsequent processing.

After satisfying the criteria established in the predicates, elements proceed to the `<Identification>` step, which marks the transformative aspect of our approach. Here, elements are mapped to specific architectural constructs using identification functions such as `toAllocation` or `toAssociation`. This step is critical to constructing a coherent view of the software architecture. The identification functions demonstrate the BNF's ability to recognize various architectural elements, from components and connectors to constructs such as subsystems.

In addition, the inclusion of `<Variable>` and `<Parameters>` in our BNF schema introduces a dynamic component to rule definitions, facilitating the maintenance and manipulation of states across iterations, conditions, and identifications. This feature enables context-aware processing of software artifacts, where one operation's results may affect subsequent operations' direction. The variable mechanism is essential for maintaining a continuous link between the original software artifacts and the derived architectural elements, thus supporting traceability throughout the reverse engineering effort. The parameters pass arguments to the various functions, allowing the reverse engineering process to be customized to meet specific requirements.

Our Retriever approach, supported by the iterative, conditional, and identification constructs within the BNF framework, enables an analysis of software systems. By allowing the association of architectural elements and supporting the thorough consideration of each relevant component within the software artifacts, our rule concept improves the

completeness and traceability of the resulting software architecture views. This view not only streamlines the discovery and analysis steps of the reverse engineering process but also enhances the view's adaptability to different ADLs.

4.2.2. Illustrative Example

This section presents an illustrative example of this rule concept, which describes the process of extracting architectural information from a software artifact developed using the Spring framework. This example demonstrates leveraging domain-specific knowledge, such as the Spring framework's annotation system, to automate the extraction and identification of architectural elements from software artifacts. By applying extraction rules that understand and interpret these technology-specific patterns, it is possible to accurately map the implementation details of an artifact to a model-based architectural view, thereby improving the understanding and documentation of software architecture views. However, the need for source code annotation is not a requirement for our approach, which allows for more in-depth analysis, including consideration of inheritance relationships. This approach streamlines the reverse engineering process and provides a structured means to visualize and analyze the architectural aspects of software systems developed with specific technologies.

```
1  @RestController
2  @Path("/notifications")
3  class EmployeeController {
4     @GET
5     @Path("/employees")
6     Set<Employee> all()
7
8     @GET
9     @Path("/employees/{id}")
10    Employee one(@PathVariable int id)
11
12    @DELETE
13    @Path("/employees/{id}")
14    void delete(@PathVariable int id)
15 }
```

Listing 4.2: Source code snippet for a REST controller with three handler methods implemented using the Spring framework.

Listing 4.2 shows a source code fragment for the REST controller `EmployeeController` that is adorned with Spring annotations such as `@RestController`, `@Get`, and `@Delete` to define three handler methods. These annotations help identify the REST controller and its associated HTTP request mappings. They encapsulate the functionality to get all employees, get an individual employee by using identification, and delete an employee.

```
1 for (c : a.declarationView())
2    if c.hasAnnotation("*Controller")
3       c.toComponent()
4       for m : invocationView(c)
5          if m.hasAnnotation("*Path")
6             m.toRole(c)
```

Listing 4.3: Source code snipped for the extraction rule to transform the technology-specific implementation.

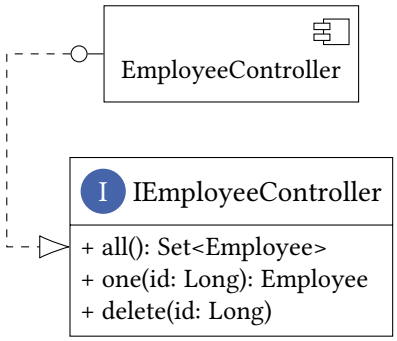


Figure 4.1.: A graphical representation of a component with its interface declaration is provided, allowing for a model-based view of the artifact to be achieved through applying extraction rules to the technology-specific implementation within an artifact. This figure is adapted from the following authored publication: [Kir+23a]

Then, Listing 4.3 introduces an extraction rule, implemented in the Xtend programming language, that aims to transform this technology-specific implementation into a model-based architectural view. This rule iterates over the classes within the given artifact, applying criteria to determine if a class is annotated as a controller (`*Controller`) and further examining the methods within these classes for HTTP mapping annotations (`*Path`). If these conditions are met, the class is identified as a component, and its methods are recognized as provided roles, facilitating the mapping of technology-specific interface implementations to a software architecture view.

Figure 4.1 illustrates this process through a UML-like graphical representation that shows the `EmployeeController` component and its declared interface, `IEmployeeController`. This diagram illustrates the implementation relationship between the component and its interface, with the interface detailing the signatures of the handler methods identified by the extraction rule.

4.2.3. Create Multiple Views with Rules

A step in the automated extraction of technology-induced views of the software architecture is analyzing the views of the artifacts by applying rules that manifest themselves as model-to-model transformations. These rules, an integral part of the process, can be defined as technology-specific, taking advantage of particular technologies' characteristics and patterns, or they can be project-specific and tailored to each project's requirements.

Our ruleset approach includes a framework designed to facilitate the accessible and reusable encapsulation of this domain knowledge.

Within this framework, predefined rules play a critical role in identifying and aggregating relevant architectural elements and their relationships among each other from the plethora of elements in the code view. These rules recognize structural patterns within the code and enable the mapping of code artifacts to their corresponding architectural components. Through precise definitions, the rules articulate the transformations necessary to isolate relevant entities and give them architectural meaning. This process enables the identification of elements of a software system, such as components, interfaces, and communication paths. It also facilitates the composition of these components into larger logical constructs that can encapsulate services or subsystems, similar to microservices.

Using queries in this framework provides curated views of the codebase and isolates elements suitable for further analysis. These elements are further examined under the conditions defined in the rules, filtering the elements to isolate those most relevant to the architectural view. The identification process describes the types of elements that manifest in the architectural view, following conventions and terminologies relevant to component-based software architectures. Beyond the mere identification of architectural elements, the rules extend to the definition of relationships between these elements to create a coherent architectural structure linked by compositional logic.

Extending the aspects of rule application, our Retriever approach includes the concept of multiple rule applications. This diversity in rule application opens up the possibility of simultaneous analysis, allowing multiple views of the same system element to be generated from different views. Such a capability is essential when a component or fragment may have various roles or functions within the system architecture, requiring its representation in architectural views.

This simultaneous processing facilitates the creation of multiple views of the same system element, with each view emphasizing different aspects or roles of the element within the overall system architecture. For example, extracting a component from its implementation view allows analysis of its internal structure and behavior, providing a view of the coding patterns, algorithms, and data structures used. This view is critical to understanding the component's functionality, performance characteristics, and possible dependencies on other software units. Conversely, viewing a component from a deployment view provides information about how it is instantiated and managed in an operating environment. This includes insight into the configuration of runtime environments, scaling mechanisms, and the component's resilience and fault tolerance strategies. Extracting a component from the view of its interactions with other calling components also reveals the nature of its interfaces, communication protocols, and data exchange dynamics. This interaction view is critical to understanding the component's role within the system architecture, including its contribution to the overall functionality of the system and its dependency on, or provision of services to, other components.

Bringing these different views into a coherent architectural model requires an approach to selecting identifiers for model elements in all views. Identifiers are essential to supporting

the traceability and coherence of model elements when merging the different views. This coherency facilitates the integration of different architectural views into a unified model. It enables architects and engineers to navigate the relationships and dependencies within the system with clarity.

Implementing such a parallel rules application requires a framework that manages the dependencies and interactions between the rules and the elements they generate. The framework we describe below for extracting structural views of a system supports cohesion and coherence between these views when multiple views of the same element are generated. This avoids conflicts and incoherence that could compromise the integrity of the architectural views.

4.2.4. View Traceability Through Link Creation

The creation of trace links is critical to extracting architectural views from artifacts. These trace links connect the derived model elements – representing different views of the artifacts – to the original artifacts. This process is integral to supporting the traceability and integrity of the transformation, thereby facilitating a coherent and accurate reconstruction of the software architecture from its constituent artifacts.

When a view of an artifact is extracted through the application of rules, each rule is inherently designed to transform the artifact into a new model element and instantiate a corresponding trace link. This trace link is critical because it explicitly denotes the origin of the model element, binding the newly created view to its source artifact. The methods specified as identifiers within the BNF play a critical role in this process. These identifiers are responsible for identifying and transforming the model element and generating the trace link, emphasizing their function in the transformation process.

The generation of trace links during the transformation process is a designed mechanism within the model-driven reverse engineering framework. When rules are applied to an artifact, creating a new model element, the identifiers inherently support that a trace link is simultaneously created. This trace link acts as a bidirectional reference, allowing navigation between the model element and its corresponding artifact. Creating trace links during the model element identification step emphasizes the granularity and specificity of the extracted architectural views. By documenting the origins of each model element within the architectural view, trace links improve the traceability of the reverse-engineered architecture. They enable stakeholders to accurately understand the derivation and rationale behind each component, interface, or other architectural construct within the model, providing a transparent and defensible mapping of artifacts to architectural elements.

The generation of trace links is intended to improve the reliability of the reverse engineering process. It enables architectural knowledge management because trace links encapsulate information about the relationship between artifacts and their architectural representations. This, in turn, helps preserve the knowledge embedded in the software system, facilitates its dissemination among team members, and supports the system's ongoing evolution.

Algorithm 4.3 Rule-based extraction of behavioral views from given trace connections between structural views and artifacts. It can be extended to extract additional aggregated behavioral views.

Input: L ▷ *Set of all structural trace links*
Input: E ▷ *Set of all extraction rules*
Output: V ▷ *Aggregated set of all extracted views*

```
1:  function ExtractBehavioralViews(L)
2:      V_p ← ExtractPerformanceViews(L)
3:      V_v ← ExtractVulnerabilityViews(L)
4:      ▷ At this point, other behavioral views can be created. ◁
5:      V ← V_p ∪ V_v                ▷ Aggregate extracted views of the system
6:      for all (v_s, a) ∈ L do      ▷ A view is linked to an artifact
7:          V ← V ∪ {v_s}
8:      end for
9:      return V
10: end function
```

4.3. Extract Behavioral Views

Due to their inherent complexity, reverse engineering software systems' performance and security properties are challenging in software engineering. This challenge is particularly pronounced when dealing with systems, where existing documentation may be fragmented or obsolete, undermining the applicability of conventional analysis techniques. In such contexts, reverse engineering emerges as an indispensable strategy, providing a way to delineate architectural views of software systems.

To address this problem, we present an automated strategy that enhances reverse-engineered architectural views by incorporating targeted insights into performance and security issues. Our approach emphasizes the need to transform code-level behavioral descriptions into abstract views that capture the essence of component interactions. Such a transformation is critical for identifying and mitigating potential performance bottlenecks and inefficiencies and providing an understanding of the system's operational behavior.

In parallel, our Retriever approach strengthens system security assessment by leveraging state-of-the-art automated tools capable of pinpointing vulnerabilities through static code analysis and dependency checking. This two-pronged analysis provides a view of potential security threats extending beyond traditional focus areas, including deployment configurations and system-wide interactions vulnerable to exploitation.

The algorithm described in Algorithm 4.3 represents our structured approach for rule-based extraction of behavioral views from software systems, leveraging established trace connections between structural views and artifacts. This process is beneficial for improving the understanding and documentation of software systems by providing insight into performance issues and potential vulnerabilities, thereby facilitating an understanding of the software's operational behavior and security posture.

At the core of this algorithm are two inputs: the set of all structural trace links L and the set of all extraction rules E. The structural trace links L serve as the foundation for the extraction process, encapsulating the relationships between the software's structural view v_s and their corresponding artifact a. In Algorithm 4.2, we described how each extracted view is associated within a trace link $a\,(v_s, a) \in L$ that records its origin and establishes a link between the structural view v_s and the artifact a from which it was derived. These links are critical for guiding the extraction process by defining the scope of the analysis. Meanwhile, the extraction rules E embody the criteria and logic for identifying and delineating the behavioral views of the structural components and artifacts.

The ExtractBehavioralViews function works by calling two sub-functions: ExtractPerformanceViews and ExtractVulnerabilityViews, which return the V_p and V_v variables, respectively. The first, ExtractPerformanceViews, focuses on extracting views that model the performance characteristics of the software system. This involves analyzing the trace links L and applying the extraction rules E to identify patterns, behaviors, and configurations that may affect the system's performance. The second function, ExtractVulnerabilityViews, identifies views highlighting potential vulnerabilities within the software system. The goal is to examine the structure and behavior of the software for security vulnerabilities or threats. This view is for proactively addressing vulnerabilities and improving the system's security posture.

After executing these sub-functions, the Algorithm 4.3 aggregates the extracted views into a set V, which represents the amalgamation of performance-related views V_p and vulnerability-related views V_v. This aggregation is further extended by a loop that iterates over each trace link $(v_s, a) \in L$, adding additional structural views v_s related to the analyzed artifacts a to the aggregated set V. This iterative process supports the idea that the extracted behavioral views encompass various insights derived from behavioral and structural considerations. Finally, the function returns the aggregated set V, which embodies a representation of the behavioral characteristics of the software system.

In the following sections, we describe the detailed techniques used to maintain these enriched views of performance and security and explain the underlying mechanisms of our Retriever approach. We also present an illustrative example highlighting our approach's applicability. Through this discourse, we aim to illustrate the capabilities of our approach to empower architects and developers to perform in-depth performance and security assessments of complex software systems. This empowerment facilitates informed decision-making, contributing to software systems' management and progressive evolution.

4.3.1. Extract Performance Views

Applying quality prediction approaches to reverse engineering models requires a representation of component behavior. To create the component behavior view from the source code view, we extend the reverse engineering approach of SoMoX proposed by Krogmann [KKR10; Kro12]. We transform the behavior descriptions in the code view

into abstract behavior at the component level. For example, the behavior view reduces all internal component actions to individual nodes. Any control flow that does not affect other components (i. e., no calls to other components) is abstracted. Thus, one component behavior view can represent behavior that spans multiple methods and classes within a component. The pseudocode shows our implementation for rule-based extraction of behavioral views for the specified artifacts, with the associated links to the previously extracted structural views.

4.3.1.1. Illustrative Example

The example of transforming Java code from Listing 4.4 into a UML version 1 activity diagram in Figure 4.2 illustrates our Retriever approach through a simple visualization of software behavior in an abstract and general way. We reused the SoMoX [KKR10; Kro12] approach to create this behavioral view from the source code view. In this view, the initial variable assignments within the Java code are combined into a single internal action, simplifying the representation of transient operations. In addition, decision processes, which initially appear as branches in the code, are modeled to show possible outcomes: either calling an external method in another component or performing an internal method call within the same component. This abstraction makes it easier to understand the code's functionality. It highlights the interactions between components and the internal flow of logic, providing a view for later model-driven analysis.

```
1 void call(int x) {
2    x += x >> 2;
3    x = -x & (x << 16);
4    if ((x % 3) == 0) {
5       external.foo();
6    } else {
7       this.bar();
8    }
9 }
```

Listing 4.4: Source code of a Java method as input to the transformation to map its behavior to a model.

4.3.1.2. Trace Links for Extracting Behavioral Views

The transformation rules are automatically applied to generate behavioral views after the trace links are created in Algorithm 4.2. The extraction rules in this step create a view of the system's behavior. For the Cartesian product of all trace links used with all existing rules, the first step is to check whether an extraction rule applies to a trace link. This predicate indicates whether the rule applies to the structural view and its associated artifact.

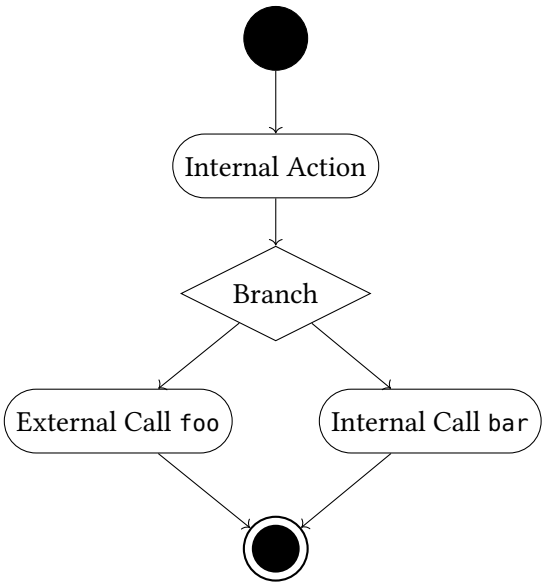


Figure 4.2.: Representation of the Java code from Listing 4.4 in a UML 1 activity diagram. The first two variable assignments have been combined into an internal action, and the branching has been adopted to lead to an external method call on another component or an internal method call within the component. This figure is adapted from the following authored publication: [Kir+24a; Kir+24b]

Algorithm 4.4 Rule-based extraction of performance views from given trace connections between structural views and artifacts.

Input: L ▷ *Set of all structural trace links*
Output: V_p ▷ *Aggregated set of all extracted views*

function ExtractPerformanceViews(L)
  $V_e \leftarrow \{\}$ ▷ *Views that provide services*
  for all $(v_s, a) \in L$ **do**
    if $e_c(v_s)$ **then**
      $V_e \leftarrow V_e \cup \{(v_s, a)\}$
    end if
  end for
  $V_p \leftarrow \{\}$ ▷ *Extended views with performance views*
  for all $(v_s, a) \in V_e$ **do**
    $V_p \leftarrow V_p \cup \mathcal{E}_p(v_s, a, V_e)$
  end for
  return V_p
end function

Algorithm 4.4 describes an approach for extracting performance views from software systems using trace links between structural views and their corresponding artifacts. The ExtractPerformanceViews function transforms code artifacts into an abstract behavioral view to examine the performance attributes of the system. This function processes the artifacts to perform actions on the structure without changing them.

At the beginning of this step, a set of structural trace links, denoted L, is required. These trace links connect the components identified in the source code to the corresponding

development artifacts, thus providing the basis for performance analysis. These components may provide services through their interfaces. Views of the internal behavior of these services in the components are extracted for later use in quality analysis, among other things. This step aims to create a set of extracted views, represented by V_p, that encapsulates the description of the behavioral aspects of all services provided within the system.

The function starts by determining a subset of views V_e containing components that provide system services. On the one hand, this subset includes the structural views that describe the components and their provided services for which a behavioral description is to be extracted. On the other hand, it also defines the calls from one component to another external component. This subset is determined by checking each trace link $(v_s, a) \in L$ against a condition e_c. This condition is used to identify the relevant views.

Once the relevant service provision views are identified, the function generates the aggregated set of service provision views, V_p. This is achieved by adding service-related information to each service view $(v_s, a) \in V_e$ by applying a transformation function $\mathcal{E}_p$. This transformation combines the information from the structural views, the associated artifacts, and the subset of service views into service views. For each view v_s, this information is successively extracted based on its associated artifacts a and inserted into the service view V_p. V_e is required to identify the external component calls the component makes in v_s.

This transformation function $\mathcal{E}_p$ performs a control flow transformation of the source code for each class or interface method corresponding to an interface signature. This transformation extends the SoMoX approach proposed by Krogmann [KKR10; Kro12] to generate the component behavior view from the source code view. This function examines each statement and, if necessary, generates an action in the behavioral view. The most important statements in this context are calls from one component to another, representing data flow and control between components. Component-internal calls, which are method calls that call a method of the same class or a method of a class within the same component, are also part of the control flow transformation. Other control flow elements, such as conditional statements, branching, looping, or iteration, are also represented in the behavioral view.

These views are then merged into the aggregated set V_p to represent the system's services. Finally, the function returns the aggregated set of all extracted service views V_p.

4.3.1.3. Control Flow Structures in Component Behavior

As mentioned above, to create the component behavior view from the source code view, we extend the reverse engineering approach of SoMoX proposed by Krogmann [KKR10; Kro12]. We have already explained the SoMoX control flow transformation in Section 2.6.3. For this section, it is relevant to know that the control flow transformation $\mathcal{E}_p$ performs a control flow analysis for each method corresponding to a component behavior to generate the actions within a behavior view from the source code view. To accomplish this, the

control flow transformation uses a two-step process. Applying these steps results in a component view consisting of the system's static structure and behaviors in the form of a behavior view.

In the first step of the process, which has already been completed, the source code was analyzed to create the trace links. The trace links contain information about how the classes, their interfaces, and methods are mapped to architectural elements such as components, interfaces, and signatures. This is used to identify external callers so that methods that provide or require external services are marked accordingly in the source code view.

In the second step, the transformation performs a control flow analysis on the source code view for each class or interface method corresponding to an interface signature. During this analysis, each statement is visited; if necessary, an action is created from the behavioral view.

The most important statements are method calls to an external component. An external call action models the invocation of a service specified in a required interface. To achieve this, it references a role that can derive the providing component and a signature that determines the called service. External call actions model synchronous calls to required services, where the caller waits for the called service to complete its execution before continuing with the actual execution. In this way, views of the behavior of different components remain independent and become interchangeable in an architectural model.

Control flow elements such as conditional statements, branches, and loops in the code view are translated into corresponding actions in the component behavior view. Conditional statements and branches such as If, Else, Switch, and Try are converted to a branch action. This branch action shares the control flow with exclusive disjunction semantics, i. e., the control flow continues at precisely one of the attached branch transitions. Loops such as (enhanced) for or while statements are converted to a loop action. This loop action models the repeated execution of its internal actions for the loop body.

Sequential internal actions are combined into a single internal action to achieve higher abstraction from the source code. This internal action combines the execution of a set of internal computations by a component service into a single model element. Computations within a component service that do not involve calls to required services are modeled this way. Internal method calls, i. e., method calls that call a method of the same class or a method of a class within the same component, are also part of the control flow analysis. Statements within an internal method call are handled as described above. Internal actions within a component can be encapsulated to preserve abstraction and avoid exposing implementation details. A high level of abstraction is necessary to keep the model accessible to mathematical analysis techniques. To achieve the desired high level of abstraction, successive internal actions are combined into a single internal action that is not mapped to other actions, such as an external call action. In principle, it is also possible to use several internal actions directly, one after the other, to model at a lower level of abstraction.

4.3.2. Extract Security Views

Detecting security vulnerabilities at the architectural stage is valuable; it allows analysis beyond source code details, potentially revealing properties related to deployment and other system-level factors. Architectural security analysis exists to help evaluate these properties.

However, building software architectures and security models remains an uphill task. This is especially true for systems, where the initial modeling effort is more extensive. While simple tools can facilitate certain aspects of the modeling process, the effort required remains a barrier. Automated recovery approaches offer potential solutions to alleviate this burden. While traditional architecture recovery emphasizes structural extraction (i. e., components), our architectural attack analysis requires additional input in the form of vulnerability information. Software architects can find this data in sources such as the U.S. National Vulnerability Database (NVD) [US 24] that provide publicly accessible interfaces. In addition, static source code analysis tools can identify vulnerabilities within components using the same classification systems as these databases.

By combining reverse engineering of the software architecture with curated security information, models are enriched with security properties. The result is a component-based architectural analysis model, explicitly focusing on analyzing security properties.

4.3.2.1. Illustrative Example

```
1 @RestController
2 public class HelloController {
3    @GetMapping("/hello")
4    public String helloWorld(World world, Model model) {
5       return "Hello " + world;
6    }
7 }
```

Listing 4.5: Source code snippet for a REST controller with a handler method implemented using the Spring framework. A REST controller created in the Spring framework exposes entry points into the software system. These entry points allow external interactions, accepting user-provided data.

We use the example in Listings 4.5 and 4.6 of a Java-based REST controller built with the Spring framework [VMw24] to illustrate our approach to automatically deriving vulnerability views. By examining source code and configuration files, this example demonstrates the multistep approach to reverse engineering software architectures with a focus on identifying and annotating vulnerabilities.

The source code snippet presented in Listing 4.5 exemplifies a REST controller that uses Spring annotations such as `@RestController` and `@GetMapping` to delineate entry points into the software system. Critical for enabling external interactions, these entry points involve accepting user-provided data, which can introduce vulnerabilities. The domain

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
   <version>2.5.4</version>
</dependency>
```

Listing 4.6: Maven build configuration fragment for a Spring 2.5.4 dependency. A Maven build configuration `pom.xml` is critical to understanding a project's external dependencies. Modern software development relies heavily on third-party libraries and frameworks. While these dependencies streamline development, they introduce an attack surface into an application. Vulnerabilities within these dependencies become inherited vulnerabilities of the software system.

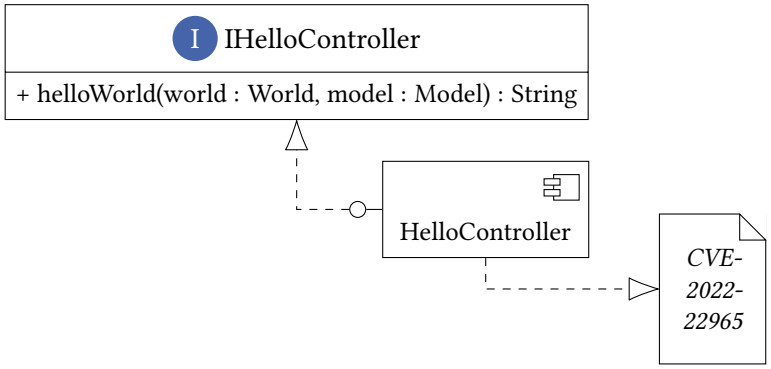


Figure 4.3.: Mapping the REST controller code from Listing 4.5 into a component with an interface definition annotated with vulnerability information based on the dependencies in the Maven build configuration from Listing 4.6. This step encapsulates the core challenge of automated vulnerability modeling. It bridges the gap between multiple representations: source code, build system artifacts, and the formal architectural model. This figure is adapted from the following authored publication: [Kir+23b]

knowledge encapsulated in these annotations helps to infer the component's interfaces, setting the stage for an in-depth analysis of the software's structural and behavioral aspects.

In parallel with source code analysis, examination of build configurations, as illustrated in Listing 4.6, reveals the software's reliance on external dependencies, specifically a Spring framework dependency defined in a Maven [Apa24b] build configuration `pom.xml`. While reliance on third-party libraries and frameworks streamlines development efforts, it also introduces potential vulnerabilities. The Maven configuration fragment highlights the importance of understanding a project's external dependencies to capture the extent of its attack surface.

Building on these findings, our Retriever approach extends to annotating components within the structural view with vulnerabilities identified through analysis of development artifacts such as source code and Maven build configurations. Using third-party static vulnerability analysis tools, vulnerabilities are identified in the project's dependencies and mapped to the structural view's corresponding components. For example, the discovery of Common Vulnerabilities and Exposures (CVE) [CVE] identifiers such as *CVE-2022-22965* [US 23; VMw22; Sny22] through static vulnerability analysis of the Maven configuration

Algorithm 4.5 Rule-based extraction of vulnerability views from given trace links between structural views and artifacts.

Input: L ▷ *Set of all structural trace links*
Output: V_v ▷ *Annotated structural view with vulnerabilities*

```
1:  function ExtractVulnerabilityViews(L)
2:      V_v ← { }.
3:      for all (v_s, a) ∈ L do
4:          V ← GetCVEs(a)
5:          for all v ∈ V do
6:              a ← GetAttackVector(v)
7:              V_v ← V_v ∪ E_v(v_s, a)
8:          end for
9:      end for
10:     return V_v.
11: end function
```

file facilitates the retrieval of vulnerability descriptions from databases such as the NVD [US 24].

CVE-2022-22965 describes a remote code execution vulnerability in *Spring MVC* and *Spring WebFlux* [VMw24] applications running on Java 9 or later. The vulnerability is due to a weak data binding process that could allow an attacker to inject and execute malicious code on the target system. This CVE has a base rating of 9.8 with a network attack vector, low complexity of access, no authentication requirements, and potential confidentiality, integrity, and availability impact [US 23; VMw22; Sny22]. This information, including the attack vector and impact, is then annotated within the security view of the model.

Figure 4.3 represents the result of this process, mapping the REST controller code to a component with an interface definition and annotating it with vulnerability information derived from the analysis of Maven build configurations. This step bridges multiple representations – source code, build system artifacts, and a formal structural view – to encapsulate the core challenge of automated vulnerability modeling.

4.3.2.2. Integrate Vulnerability Analysis With Structural Views

The process described here, illustrated by the accompanying algorithm (Algorithm 4.5), outlines an approach for integrating vulnerability findings into architectural models, enabling an understanding of potential security risks. Our approach integrates vulnerability analysis directly into structural views by annotating them with vulnerabilities identified through the analysis of development artifacts.

The next step in this process is using structural trace links L, which are critical in associating software components identified through static analysis with their corresponding development artifacts. These artifacts, which include source code and configuration files, serve as the data source for vulnerability detection and analysis. Using specialized third-party software, the analysis targets Java projects that use popular build systems such as

Gradle [Gra24] or Maven [Apa24b] that are prevalent in modern software development practices.

The GetCVEs function extracts vulnerabilities from development artifacts by performing static analysis on those artifacts to generate a list of vulnerabilities, each represented by a CVE identifier [CVE]. While used by third-party static vulnerability analysis tools, this step is inherently limited by the nature of static analysis, which identifies known vulnerabilities within software dependencies. While this approach is for highlighting potential vulnerabilities within open-source dependencies or container images, its accuracy is limited by its reliance on pre-identified vulnerabilities and potentially missing vulnerabilities within larger, multi-component systems.

Each identified CVE number is further researched in the NVD using the GetAttackVector function to enrich the vulnerability analysis. The NVD details vulnerabilities, including their potential attack vectors, the privileges required to exploit them, and the expected impact on system attributes such as availability, confidentiality, and integrity. This enriched data set provides the basis for an attack propagation analysis, allowing for a deeper exploration of how vulnerabilities might be exploited in real-world scenarios.

Finally, the $\mathcal{E}_v$ function creates a behavioral view that annotates the vulnerabilities and their attack vectors with the associated structure. This view can then be used to place the vulnerabilities in the software architecture context and assess the potential impact on the integrity and availability of the system.

This process involves mapping vulnerabilities to specific software components within the structural view. Build configurations, such as Gradle's `build.gradle` or Maven's `pom.xml` files, are initially associated with vulnerabilities. While these configurations provide a high-level view of the project's structure, they do not offer direct insight into the specific code components affected by the identified vulnerabilities. A mapping process is employed to fill this gap and ensure that the vulnerability analysis reflects the potential security risks within the software system.

4.3.3. Extension Point for Additional Behavioral Views

Extending the Algorithm 4.3 to a range of behavioral views represents an opportunity to deepen our understanding of system dynamics, especially when runtime behavior is considered in addition to static analysis. As noted in the previous discussion, extending the algorithm at the point of interest opens avenues for integrating additional techniques that provide insightful views into the operational aspects of the system under study. This subsection considers the potential strategies and technologies that can be used to enrich the behavioral views extracted from software systems, mainly through runtime artifacts.

A promising approach is integrating performance views derived from log files, such as those generated by monitoring tools, during the software system's operation. Log files with measurement data provide a temporal snapshot of system performance, capturing metrics that can be critical for identifying inefficiencies, bottlenecks, and potential areas

for optimization. We can construct performance views by analyzing these logs, which provide a picture of system behavior under various conditions and workloads.

A tool in this context is the Performance Model eXtractor (PMX), developed by Jürgen Walter [Wal+17; Wal19]. PMX is meant to automate the extraction of architecture performance models from measurement data, streamlining the process of generating performance views. This tool is designed to support logs from the *Kieker Monitoring Framework* [Hoo+09; HWH12], or those compatible with the open-source *OpenTelemetry* framework [Clo24], covering a wide range of input formats and supporting applicability across different software ecosystems.

PMX's integration with our Algorithm 4.3 involves parsing the collected logs to identify performance-related events and metrics. These include response times, resource utilization, throughput, and error rates. We can construct performance views that reflect the system's runtime behavior by mapping these metrics to the corresponding components and interactions within the software architecture. This process increases the granularity of our behavioral views and provides a dynamic view that complements the static information derived from source code and configuration files.

The assembly and refinement of these performance views within Algorithm 4.3 follow a similar pattern to the behavioral views extracted from static information. Each performance view, once generated, is associated with a trace link that identifies its origin and context within the overall architecture. This association supports the idea that the performance views are seamlessly integrated into the aggregated behavioral views, enriching the architectural model with valuable insight into the system's operational performance.

5. View Composition and Refinement

The model-driven composition and refinement step improves the integration of model-providing processes, such as reverse engineering and extracting views from various software artifacts. It also integrates model-consuming processes, such as quality prediction for software systems. This step is critical in transforming and manipulating model-driven knowledge through an automated metamodel-independent framework designed to accommodate different target metamodels.

Three interrelated processes are central to the model-driven composition and refinement step: information discovery, information processing, and finalization. The process begins with information discovery, which extracts views that encapsulate different views of system elements from different artifacts. These include source code and configuration files and build- and deployment-related artifacts. Despite these views' heterogeneity and lack of coherence, the model-driven composition and refinement framework attempts to create a coherent overall view through composition and rule-based refinement, considering the predictive quality attributes through the relationships between different views.

The composition and refinement step is highlighted by its applicability, facilitated by utilizing General-Purpose Languages (GPLs) such as Java and Xtend [Bet16; Ecl24d] and transformation languages such as QVT [Kur08; QVT] and ATL [JK06; Ecl23] to define and implement refinement rules. These rules are critical for integrating isolated elements and views into a single, integrated model, revealing previously overlooked connections between views, and improving the model's predictive capabilities for quality attributes.

In addition, this step introduces interfaces that streamline the interaction between the extraction of views and their consumption. This facilitates the ingestion of information for internal processing in the discovery step and supports the delivery of processed views to end users in the finalization step. The information processing step bridges these interfaces, focusing on aggregating and refining elements and their relationships into a model.

To bridge the gap between model-supplying and model-consuming processes, we have developed an approach to facilitate the automated transformation of views derived from software systems. Algorithm 5.1 shows the processing pipeline for composing and refining the previously extracted views. This approach is metamodel-independent, shows applicability across different target metamodels, and uses GPLs such as Java and Xtend alongside transformation languages such as QVT or ATL to formulate refinement rules.

From the initial extraction of views to the final steps of model finalization, the model-driven composition and refinement step follows a holistic approach. This structured and rule-based framework facilitates an understanding of software systems. It also paves

Algorithm 5.1 The composition and refinement function includes three core steps: discovery, composition, and rule-based refinement. It extracts relevant architectural information, integrates these elements into a cohesive view representation, and applies refinement rules to improve the view's granularity, coherence, and alignment with domain-specific requirements.

Input: V_e ▷ *Set of extracted views*
Input: R ▷ *Set of predetermined refinement rules*
Input: F ▷ *Set of predetermined finalization rules*
Output: M_f ▷ *Final output models ready for consumption*

```
1: function CompositionRefinement(V_e, R, F)
2:     v_q ← { }
3:     for all e ∈ Discover(V_e) do
4:         v_q ← ProcessElement(v_q, e, R)
5:     end for
6:     return FinalizeViews(v_q, F)
7: end function
```

the way for insightful and predictive analysis of software quality attributes, bridging the gap between model-driven quality prediction and the nature of software artifacts. In this approach, model-driven composition and refinement support predictive, quality-oriented analysis of software systems by seamlessly integrating with model-driven quality prediction. It transforms raw extracted views into formats suitable for quality analysis while maintaining a balanced step that accommodates input diversity and supports the reuse of processed models.

The following sections formally describe the three steps of the proposed framework, using pseudocode for clarity. First, the process begins with information retrieval, where relevant data is collected. This step serves as a foundation that facilitates subsequent operations. Next, the framework processes elements and relationships, integrating view composition and refinement. This step involves arranging and refining elements and their relationships, supporting coherence and efficiency. Finally, the process culminates in finalization, where the assembled components are prepared for the final software architecture model for deployment or further iterations.

> The subsequent sections are based on the following authored publications: [SK19; Gst+24; Kir+24a; Kir+24b]

5.1. Information Discovery

The first step of our model-driven composition and refinement framework, the discovery step, is designed to address the challenge of integrating heterogeneous views extracted from software systems. This process, known as information discovery, is the interface between the composition and refinement step and the various sources of extracted views,

such as artifacts from previous steps. The core objective of this step is to process and consolidate the extracted views, referred to here as V_e, into a new, unified set of views, referred to as v_q. This transformation is critical to transforming potentially incoherent representations of the system architecture into a coherent set ready for further analysis and integration processes.

To manage the information provided by view extractions, we categorize the extracted information into two types: elements and relations, collectively referred to as entities. Elements are defined as atomic pieces of information. At the same time, relations are directed links between two entities, distinguishing between atomic relations (direct links between elements), hierarchical relations (relations that contain other relations), and pseudo-relations (relations that are necessary for processing but not present in the target metamodels). The discovery process involves three tasks to facilitate this categorization and integration:

1. Retrieval of heterogeneous information from view-providing processes tailored to the specific units of the processes used. This task supports the collection of views for subsequent processing.
2. Classification of the retrieved information into individual elements and relations through information decomposition is followed by translating these entities into the consumer domain (information translation or classification). This step uses knowledge of the source and target metamodels to support accurate representation and integration.
3. Provide the classified entities and relations to subsequent processes via well-defined interfaces or access points, using wrappers to minimize coupling to the specific metamodel. These wrappers provide functionality to check for equivalence and identify entities or relations as placeholders, facilitating integration and analysis.

By performing these tasks, the discovery step bridges the initial extraction of architectural views with the steps of architectural modeling. It supports the alignment of foundational views and prepares them for model-driven analysis and refinement. The model-driven composition and refinement framework addresses integrating heterogeneous views through this process. It lays the groundwork for a seamless transition to subsequent model-driven composition and refinement steps.

The Discovery function described in Algorithm 5.2 plays an integral role in the unification of elements derived from different extracted views, denoted as V_e. This function is essential to ensuring that these elements are seamlessly integrated into the subsequent steps of the overall analysis framework, facilitating a transition to a unified view, symbolized as v_q. Starting with an empty set for v_q, the Discovery function iterates through each element, e, within the set of extracted views, V_e. Each iteration involves retrieving heterogeneous information from V_e, which inherently acknowledges the diversity and disparity of information within the initial set of views. Next, each element e undergoes a classification and translation process by calling the ClassifyWrapper function. This step classifies and translates the information in e into a format for the target analysis framework. The result of this classification and translation process is then fused with v_q, progressively building a

Algorithm 5.2 The Discovery function is responsible for extracting heterogeneous elements from previously extracted views, classifying them, and translating them into the target format for making them available to subsequent processes in a unified manner. This supports integration into the overall analysis framework.

Input: V_e ▷ *Set of extracted views*
Output: v_q ▷ *New unified view of elements*

```
1: function DISCOVER(V_e)
2:     v_q ← { }
3:     for all e ∈ V_e do                    ▷ Retrieve heterogeneous information
4:         v_q ← v_q ∪ CLASSIFYWRAPPER(e)    ▷ Classify and translate information
5:     end for
6:     return v_q
7: end function
```

new, unified view that encapsulates the refined essence of the originally extracted views. Upon completion of this iterative process, the DISCOVERY function culminates by returning v_q, which now represents a coherent, unified view of the elements initially scattered across the heterogeneous set V_e.

5.2. Element and Relationship Processing

The task of composing and refining architectural elements in the software architecture domain is facilitated by the element and relationship processing step, a cornerstone of the composition and refinement step. This step serves as a central conduit from the initial identification of elements and relationships to their final assimilation into the software architecture model. This emphasizes a process that begins with integrating newly identified elements into an established collection through a method known as composition. This initial step is critical to establishing the structural foundation necessary for the subsequent refinement step.

Once the composition is complete, the step enters a refinement step, in which each element is examined according to predetermined refinement rules. The essence of the refinement step is its ability to generate additional model elements, called implications, as a direct consequence of applying refinement rules to the initial elements. These implications, which embody either dependencies or emergent elements that require further processing, play a role in the iterative improvement and extension of the architectural model.

Encapsulation within wrappers is employed to maintain metamodel independence and streamline the processing of identified elements and relationships. This level of abstraction supports the idea that elements can be processed without direct reliance on specific metamodel elements and relations, thereby extending the applicability of the step across different metamodels. The entities processed this way converge on an intermediate view, a metamodel-independent repository that facilitates entities' addition, retrieval, and recursive replacement. This intermediate view emerges as a facet of the composition

Algorithm 5.3 The process element function applies the refinement rules to the given element, generating a set of implications, and then recursively applies the same rules to each newly generated implication. This recursive approach supports the idea that refinement effects are propagated throughout the model to transform and generate a refined representation.

Input: v_q ▷ *Unified view of elements*
Input: e ▷ *Element to which rules are applied*
Input: R ▷ *Set of predetermined refinement rules*
Output: v_q ▷ *New refined view of elements*

```
function ProcessElement(v_q, e, R)
    v_q ← C(v_q, e)
    for all r ∈ R do
        for all i ∈ R_r(v_q, e) do          ▷ Applying refinement rules to elements
            v_q ← ProcessElement(v_q, i, R)  ▷ Processing implications for further refinement
        end for
    end for
    return v_q
end function
```

and refinement step, enabling the orderly aggregation and manipulation of architectural entities.

This entity processing bifurcates into two actions: composition and rule-based refinement, which can generate implications. Unlike directly processed entities, implications are dependent on and associated with a separate set designated for subsequent processing, rather than being assimilated within the same operational unit.

After processing the discovered entities and their implications, the resulting enriched intermediate view is prepared for finalization. This approach supports integrating and combining new elements within the pre-existing architectural framework. It emphasizes the cohesiveness of the evolving software architecture model. This strategy exemplifies the structured approach inherent in the composition and refinement step, central to the articulation and progressive refinement of software architecture models, thereby facilitating an accurate reverse engineering process.

The ProcessElement function, as described in the algorithm, is for applying a set of predetermined refinement rules, denoted R, to a given element e within a unified view of elements, v_q. This function is a cornerstone of the refinement process, which aims to improve the detail and coherence of the model through the application of rules and the subsequent generation and processing of implications. The function is initiated by integrating the element e into the unified view v_q through a composition operation, symbolically represented as $C(v_q, e)$. This step supports that e is considered part of the unified view before any refinement rules are applied. After this initial integration, the function iterates over each rule within the set R and applies these rules to the element e, symbolically represented as $\mathcal{R}_r(v_q, e)$. This application of refinement rules is designed to identify and generate a set of implications, each denoted by i, that result from the interaction of the

rule with the element in the context of the unified view. These implications represent new elements or modifications necessary to refine e according to the criteria specified by the rule. For each implication i generated by the application of the rule r, the ProcessElement function is called recursively. This recursion allows i to be processed in the same way as the initial element e, applying the set of refinement rules R to support the idea that the implications themselves are refined and integrated into the unified view v_q. This recursive approach supports the idea that the effects of refinement are propagated throughout the model, enabling a transformation and generating a refined representation of the model. Upon completing this iterative and recursive process, the function returns the new refined view of elements, v_q, which now includes the initially provided element e and all implications generated and refined throughout the process. This process exemplifies an approach to model refinement that supports the idea that each element and its associated implications are processed according to the predetermined refinement rules, culminating in a refined and improved view of the model.

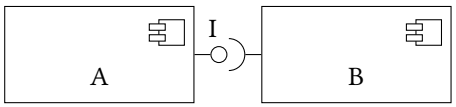
5.2.1. View Composition

The step of the view composition serves as an approach to constructing a view of a software system. This process begins with the definition of v as the pre-existing view for all model elements and e as a new model element not originally part of v, e. g., $e \notin v$. Critical for this integration is the requirement that both v and e adhere to the same metamodel, thereby supporting their inherent compatibility and the overall coherence of the system architecture.

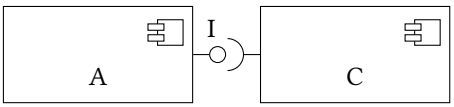
Model composition, represented by C, is designed to assimilate e into v, provided that e does not duplicate any element in v. This assimilation promotes the creation of a new, enriched view $v' = C(v, e)$ that encapsulates both the original and the newly added elements. The operation goes beyond simple addition by employing a process of identification and merging, where model elements with identical identifiers are consolidated into a single entity. This approach reduces redundancy and promotes a unified system view.

Establishing the identity of model elements across different types is essential for combining elements. This necessitates a precise definition of equality specific to each element type to support an accurate view composition. For example, classes are typically distinguished by their qualified names, incorporating the namespace and the class name. This is exemplified by the identifier `com.example.myproject.MyClass`. In contrast, instances are identified by their associated class and an identifier, such as a variable name. It is common for interfaces to be determined by URIs, such as `https://example.com/myproject/MyInterface`, providing a universal identification method. Similarly, configuration files are frequently identified using their file paths. This definition of equality is essential for accurately composing model elements within software architecture views.

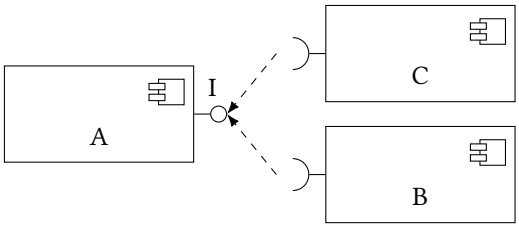
In addition, the composition process extends to the relationships between model elements, evaluating and merging these connections based on their specific types and the



(a) This diagram illustrates a UML assembly connector that represents a simple dependency between components A and B through the required interface I.



(b) Before integrating the AB relationship, this view shows the initial interactions between components A and C, connected by the shared interface I.



(c) After the composition process, this enriched view encapsulates components A, B, and C, along with the newly introduced AB relationship and the original AC relationship. This representation emphasizes the increased interconnectivity within the system that results from composition.

Figure 5.1.: This graphical representation shows the composition process in which the AB relationship is introduced into the existing configuration of components A and C connected by AC. This integration results in a new, unified view of three interconnected components that capture the essence of view composition in software architecture. This figure is adapted from the following authored publication: [Kir+24a; Kir+24b]

identities of the referenced elements. Integrating new model elements and their relationships transforms a connected system view that reflects the interdependencies in software architectures.

This composition activity is embedded within a larger framework of entity processing that includes the task of information merging or composition. This step supports incorporating an entity into the intermediate view and manages the resulting dependencies. Given the model's atomic view of entities as indivisible units of knowledge, the concept of dependencies is redefined. In particular, integrating entity relationships requires a recursive approach to include child entities in the implications. This mechanism supports a structured processing sequence for entities that adhere to a logical hierarchical order, thus preserving the structural integrity of the view.

Our concept of integrating new elements into existing models is illustrated in Figure 5.1, which shows the procedural addition of a new relationship between components within an architectural view. This example serves as a concrete demonstration of the composition process in software architecture modeling. The model evolves by adding and integrating new relationships into existing views to reflect the interdependencies among its components.

Figure 5.1a begins this illustration by showing a relationship between two components, A and B. In this simplified scenario, component B requires an interface I provided by component A, thereby establishing a direct dependency relationship AB. This scenario represents an assembly connector in UML that shows the association between these components through the required interface.

Expanding on this basic setup, Figure 5.1b introduces a third component C, which requires an interface I from component A before integrating the relationship AB. This view lays

the groundwork by highlighting the pre-existing relationship AC between components A and C, setting the stage for the subsequent addition of a new relationship.

The result of this process is shown in Figure 5.1c, where integrating the new relationship AB into the view containing components A and C, along with their existing relationship AC, creates a new view. This enriched view combines components A, B, and C, along with their respective relationships, AB and AC, and the standard interface, I. In particular, the components A shown in Figures 5.1a and 5.1b are identified as identical entities and consequently merged into a single component A in Figure 5.1c, supporting a coherent and unified representation of the software architecture.

5.2.2. View Refinement

The improvement and accuracy of software architecture models through reverse engineering are anchored in model refinement and rule-based entity processing. This approach aims to refine extracted views by adding detail, thereby creating models at different levels of abstraction that are linked together through refinement. This step is essential to developing new model elements, called implications, which are critical to supporting the idea that the evolving model obeys the predetermined refinement rules and the constraints of the target metamodel.

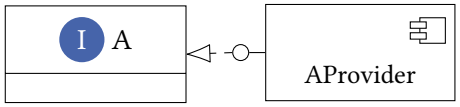
Central to this step are refinement rules, conceptualized as model-to-model transformations, whose task is to adjust the current model view v to conform to rules. These rules, denoted by r and targeting specific model elements e, are operationalized by the transformation $v' = \mathcal{R}_r(v_q, e)$, where v' represents the result of the refinement process. The function of these rules is to evaluate the compliance of each model element with the specified conditions and to generate and incorporate necessary implications into the model when an element fails to satisfy a rule, thereby supporting model compliance.

The essence of refinement rules lies in their ability to dictate that output views must conform to the requirements derived from the target metamodels or the requirements of the subsequent model consumption step, thereby supporting model integrity and compatibility. Thus, the refinement effort goes beyond mere adjustments to the intermediate view to include placeholder entities. These entities are replaced or redefined through direct or indirect strategies to maintain model accuracy without relying on false premises.

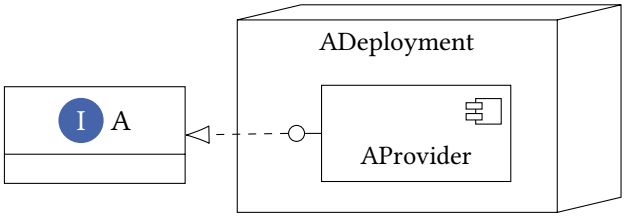
Direct placeholder replacement occurs when a placeholder relation is replaced with a concrete entity determined by the unity of its child entities. Alternatively, indirect replacement adjusts the composition of relations by modifying child entities as required by their placeholder or token status, thereby supporting an authentic representation of the intended architectural configuration.

In addition, this approach includes a recursive mechanism for processing entities and their resulting implications. This cyclical adaptation allows the model to evolve under newly introduced or modified rules, addressing placeholder entities to satisfy the requirements of the refinement rules. This process manages entities affected by replacements within a

(a) Simplified representation of an arbitrary view after composition of an interface A. No component provides the interface. Therefore, the refinement rule r_1 is not satisfied, and the refinement process is started.



(b) The view after the composition of interface A and refinement for refinement rule r_1. The refinement results in an implication in the form of a component AProvider that provides interface A. After the implication is composed, the view satisfies the refinement rule r_1 but results in unsatisfied refinement rule r_2.



(c) The view after the composition of interface A, the refinement for refinement rule r_1, and the refinement for refinement rule r_2. The view satisfies all the refinement rules. As a result, the recursive refinement process that was initially started by the composition of interface A is terminated.

Figure 5.2.: Graphical representation of an example of recursive view refinement using two example refinement rules r_1 and r_2. The r_1 refinement rule specifies that each interface must be provided by at least one component. The r_2 refinement rule specifies that a deployment node encapsulates each component, and its interfaces are delegated through the encapsulating deployment node. This figure is adapted from the following authored publication: [Kir+24a; Kir+24b]

provisional view, ensuring that their successors are correctly processed as implications, thereby incrementally improving the fidelity and completeness of the model.

The combination of model refinement techniques and rule-based processing outlines a strategy for navigating the intricacies of software architecture model development. This strategy supports the integration of improved details into the architectural framework. It guarantees that the model conforms to the metamodel's high-level structural guidelines and specific constraints, enabling thorough and accurate reverse engineering.

The refinement process in the domain of automated reverse engineering of software architecture models is illustrated by a UML-like example in Figure 5.2. It shows the application of two refinement rules – denoted r_1 and r_2 – in the context of transforming an arbitrary architectural view V into a compliant model that satisfies both specified rules. This iterative process demonstrates the application of model-to-model transformations critical to achieving a refined software architecture model that conforms to predefined architectural standards and constraints.

The initial state of the architectural view, represented as V, results from the composition of an interface A with no providing component, making the model non-compliant with the refinement rule r_1. The r_1 rule requires that each interface present in the model must be provided by at least one component for a view to be considered refined by its criteria. The absence of a component providing the A interface signals the need to apply a refinement to bring the view into compliance with r_1. The intermediate state of the architectural view after refinement, shown in Figure 5.2b, illustrates the transformation

of V into V' through the integration of an implication, a component named `AProvider` that satisfies the provisioning requirement for the A interface, thereby satisfying r_1. If no further information about the component `AProvider` can be obtained from the other views, it is completed with a minimal behavior description. This behavior description would contain only an empty SEFF for the methods defined in the interface A.

However, the refinement journey does not end with the satisfaction of r_1 because the model V' now faces the non-compliance of another refinement rule, r_2, which requires that a deployment node encapsulate each component. In addition, r_2 specifies that the encapsulating deployment node must delegate the exposed interfaces of encapsulated components to achieve rule compliance. The subsequent refinement step, illustrated in Figure 5.2c, extends V' to V'' by incorporating the necessary implications to satisfy the requirements of r_2 – namely, the addition of an encapsulating deployment node and the delegation of the interface A by that node.

The result of the refinement process is embodied in the view V'', a model that harmoniously satisfies the conditions of both r_1 and r_2 refinement rules. This final view represents a refinement of the original architectural representation, with the sequential application of implications and recursive implications mandated by the refinement rules. The recursive nature of this process emphasizes its ability to iteratively incorporate necessary adjustments and implications until all refinement rules are satisfied. The process ends with achieving a coherent and compliant model with the architectural standards described by the refinement rules.

The example shown in Figure 5.2 emphasizes the principles and mechanisms underlying the recursive refinement process and provides a concrete illustration of how potentially incoherent representations of a software system's architecture can be transformed into a coherent and compliant model. This exemplification illustrates the application of refinement rules in automated reverse engineering and confirms the role of iterative refinement in creating precise and accurate software architecture models.

5.3. View Finalization

The finalization process marks the final step of the composition and refinement step. It bridges the model-driven composition and refinement framework and various view consumers, such as tools or approaches focused on model-based quality prediction. This final step is essential for translating the refined and intermediate views processed through earlier composition and refinement steps into one or more output views downstream applications or processes can easily consume.

Dedicated finalization units, each tailored to a specific output view, are integrated into the finalization process. These units perform two functions critical to transforming the intermediate view into its final form. The first function, information mapping, involves identifying and extracting relevant entities from the intermediate view. This step requires

Algorithm 5.4 The finalization function transforms the intermediate view into output views tailored for various view consumers, such as tools or approaches focused on model-based quality prediction, supporting semantic and syntactic compatibility with consumer expectations.

Input: v_q ▷ *Intermediate view after composition and refinement*
Input: F ▷ *Set of predetermined finalization rules*
Output: M_f ▷ *Final output models ready for consumption*

```
1:  function FinalizeViews(v_q, F)
2:      M_f ← { }
3:      for all f ∈ F do
4:          v_f ← { }
5:          for all e ∈ v_q do
6:              if f(e) then                    ▷ Identify relevant elements
7:                  v_f ← v_f ∪ e               ▷ Add to output view
8:              end if
9:          end for
10:         M_f ← M_f ∪ F_f(v_f)                ▷ Add to output view
11:     end for
12:     return M_f
13: end function
```

understanding the intermediate view's structure and the target domain's requirements to support the idea that only relevant entities are selected for finalization.

After retrieving the relevant entities, the finalization unit takes on the task of view generation. This involves transforming the selected entities into a format that meets the specific needs and standards of the target domain. View generation is not simply a process of translation. Still, it involves recontextualizing the entities to support that they are syntactically and semantically compatible with the expectations of the target domain. This supports the idea that the resulting views can be seamlessly integrated into view-consuming processes. This facilitates transitioning from the model-driven composition and refinement framework to applications in quality prediction or other domains.

After completing these tasks, the transformed views are available to the view-consuming processes. The model-driven composition and refinement `Transformer` plays a role in this step, acting as a facilitator that bridges the gap between the processes of model refinement and the needs of consumers. This process encapsulates the essence of the composition and refinement approach, highlighting its ability to refine and assemble views and tailor those views to specific consumer needs. This improves the applicability of automated reverse engineering in software architecture models.

The FinalizeViews function, as described in Algorithm 5.4, transforms the intermediate views, denoted v_q, into final output views, v_f. These views are prepared for consumption by various model consumers. These consumers may include tools or approaches focusing on model-based quality prediction, requiring that the output views support semantic and syntactic compatibility with the consumer's expectations. Starting with the intermediate

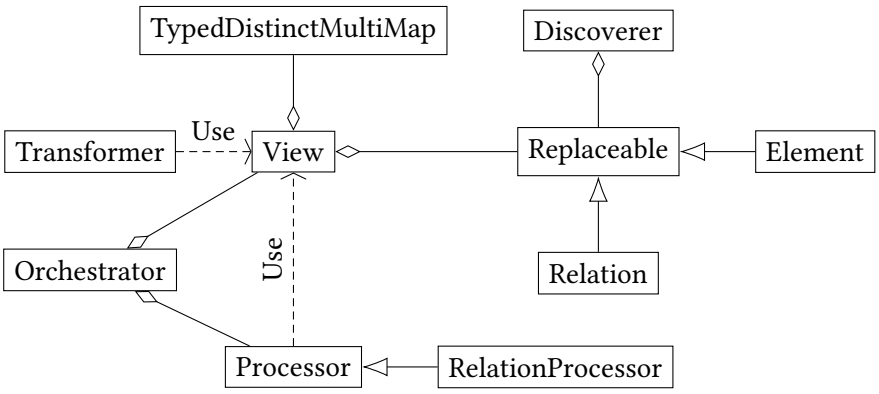


Figure 5.3.: The `TypedDistinctMultiMap` class provides a basic structure for type-safe storage and retrieval, which is critical for organizing model data based on type keys and facilitating input management. The `Discoverer` class, integral to the initial steps of model processing, specializes in storing and categorizing replaceable instances, using type-specific methods to manage data. The `Element` and `Relation` classes extend the functionality of replaceable instances, with `Element` focusing on individual model components and `Relation` emphasizing the connections between them to support complex model architectures. The `Processor` and `RelationProcessor` classes are designed to perform specific data processing tasks on the model elements and relationships, respectively, with `RelationProcessor` providing additional capabilities to handle the intricacies of relational data. Finally, the `Orchestrator` plays a key role in managing the workflow between these components, coordinating the flow of data from discovery through transformation, encapsulated by the `Transformer`, which transforms intermediate models into consumable formats for various applications. This figure is adapted from the following authored publication: [Gst+24]

view v_q, which has undergone previous steps of composition and refinement, the function initializes an empty set for v_f, representing the nascent state of the final output view. The function then iterates over each element e within v_q, evaluating to determine the relevance of each element to the finalization process. This evaluation is symbolized by the condition $f(e)$, a predicate function that determines the applicability of the element e to the construction of v_f. After identifying an element e as relevant, the function uses a transformation or formatting operation, $\mathcal{F}(e)$, to adapt e into a form that is congruent with the target domain of the view consumers. This adapted element is then merged with v_f, progressively enriching the output view with elements tailored to the needs of the view consumers. The iterative process culminates in aggregating all relevant and appropriately transformed elements into v_f, which is then returned as the final output view for further consumption.

5.4. Metamodel-Independent Framework

The model-driven composition and refinement step is embodied in an open-source, metamodel-agnostic framework that incorporates the principles of metamodel independence. This framework is built using the principles of object-oriented programming with Java as the programming language of choice, leveraging Java's platform-independent capabilities and widespread acceptance in the professional community.

The structural blueprint of the model-driven composition and refinement framework, shown in Figure 5.3, corresponds to the basic process structure that is integral to the model-driven composition and refinement approach. This structure is operationalized by developing three central process-related components within the framework: the `Discoverer`, the `Processor`, and the `Transformer`. These components, or their classes, embody the core functionalities of discovery, processing, and transformation inherent in the model-driven composition and refinement step. An orchestrator component has been introduced to complement these components to facilitate seamless interaction and coordination between the `Discoverer`, `Processor`, and `Transformer`. In addition, a `Surrogate` component is incorporated to encapsulate the intermediate view and serve as a central repository throughout the transformation process.

A feature of the model-driven composition and refinement framework is its hourglass process architecture, reflecting its metamodel-independent implementation approach. This configuration emphasizes the abstract and extensible nature of the framework regarding the discovery and transformation steps, in contrast to the concrete approach adopted for the information processing step. The design philosophy behind this hourglass shape is to produce diversity at both ends of the process spectrum – discovery and transformation – to support high adaptability to a wide range of input and output formats. Conversely, the narrow focus on information processing encourages the reuse of metamodel-independent functionality, optimizing the framework's applicability across different modeling environments.

This architecture facilitates adaptability in handling input and output information. It maximizes the reuse of core functionality, thereby striking a balance in model-driven composition and refinement processes. Through this architecture, the model-driven composition and refinement framework sets a new benchmark for facilitating the seamless integration of model-providing and model-consuming processes, contributing to software engineering and model-driven development.

5.4.1. Surrogate

The `Surrogate` component, an integral part of the model-driven composition and refinement framework, encapsulates a set of classes and interfaces designed for the dual purpose of information classification and caching. Central to its function is using a construct called "replaceable", which is central to information classification. In this context, each discovered element or relationship is encapsulated within an instance of `Replaceable`, providing a structured approach to information management within the framework.

A `Replaceable` instance can determine whether it represents a placeholder through the `isPlaceholder` method, a feature for distinguishing between definitive and provisional elements. In addition, these instances have the functionality to suggest potential replacements via the `replace` method, facilitating refinement and evolution of the view.

In addition to the generic `Replaceable` construct, the model-driven composition and refinement framework enriches its functionality by providing specialized implementations

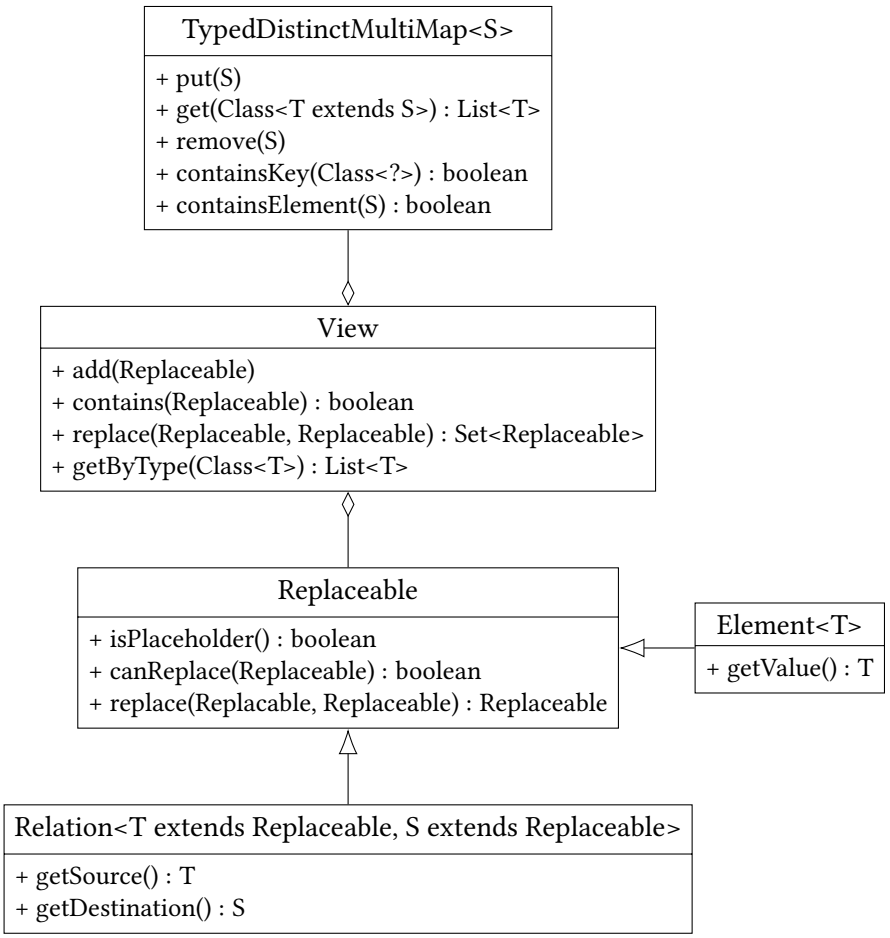


Figure 5.4.: The `TypedDistinctMultiMap` implements a type-safe storage system where each entry is unique, using types as keys to organize and retrieve data. The `View` class facilitates the management and presentation of software architecture views, enabling structured visualization and interaction within the framework. The `Replaceable` class provides mechanisms for identifying whether an instance is serving as a placeholder and for suggesting potential replacements, thus supporting view updates. The `Relation` class encapsulates relationships between elements and provides methods to access the source and target entities, supporting relationship management within views. The `Element` class wraps individual view elements and provides access to the underlying data through a method that returns the wrapped element, improving data encapsulation. This figure is adapted from the following authored publication: [Gst+24]

designed for wrapping atomic elements (`Element`) and relationships, whether atomic, hierarchical, or pseudo-relational (`Relation`). The `Element` wrappers provide the `getValue` method to retrieve the underlying element. In contrast, the relation wrappers provide the `getSource` and `getDestination` methods to access the source and destination entities, respectively. This improves the framework's ability to model relationships within the software architecture.

The intermediate `View`, a cornerstone of the model-driven composition and refinement approach, is designed to be metamodel-independent, supporting the addition (`add`) and retrieval (`getByType`) of elements and relations as interchangeable instances. This design supports type safety, which is critical to maintaining view integrity and accuracy. The

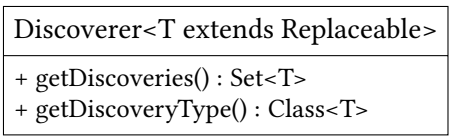


Figure 5.5.: The `Discoverer` class manages the collection and retrieval of interchangeable instances within the framework, allowing them to be categorized and accessed according to their specific types. This figure is adapted from the following authored publication: [Gst+24]

framework uses a `TypedDistinctMultiMap` for type-safe storage and retrieval, using types as keys and supporting that each entry is unique. This multimap facilitates the management of view content, including the ability to check for the presence of certain replaceable instances (`contains`) and replace them as needed, thereby recursively updating the view with their successors (`replace`).

The `TypedDistinctMultiMap` class is designed for type-safe storage and retrieval, using types as keys to maintain a set of elements. The `put` method allows an element to be added to the multimap. Element retrieval is facilitated by the `get` method, which returns a list of elements of a specified type. The `remove` method allows a specific element to be removed. The class also provides `containsKey` to check the presence of a type as a key and `containsElement` to check if an element exists in the map. Together, these methods provide an entire class for managing collections that require type distinction in their elements.

Through these mechanisms, the `Surrogate` component contributes to the applicability of the model-driven composition and refinement framework, allowing the handling of different types of information. This supports the view's evolution, which aligns with the underlying metamodel-independent principles.

5.4.2. Discoverer

The initial discovery step is encapsulated by the abstract `Discoverer` class within the model-driven composition and refinement framework. This class is specifically designed to manage and encapsulate a collection of exchangeable instances, allowing these instances to be stored and retrieved through the `getDiscoveries` method. In addition, the `Discoverer` class can identify and return the specific type of interchangeable instance it is designed to handle, as facilitated by the `getDiscoveryType` method. This distinction supports that each `Discoverer` instance is associated with a single, specific type of exchangeable instance within the model-driven composition and refinement framework. This emphasizes the framework's structured approach to managing the discovery and initial classification of elements in a model-driven development process.

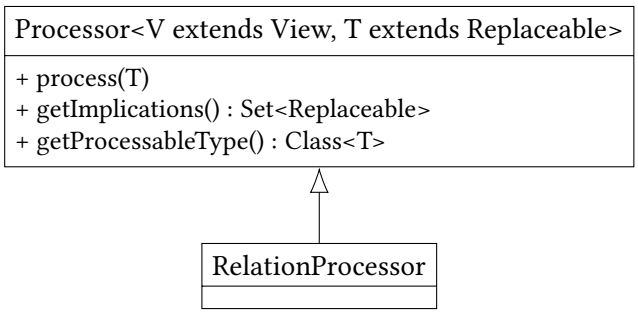


Figure 5.6.: The `Processor` class implements basic view element composition and refinement functionality, serving as a tool for transforming view data. Building on the Processor class, the `RelationProcessor` specifically handles generic relationship processing and includes placeholder replacement logic that facilitates detailed and precise view refinement. This figure is adapted from the following authored publication: [Gst+24]

5.4.3. Processor

Specialized processors perform the information processing in the model-driven composition and refinement framework. These processors can perform two operations: composition and refinement of entities within an intermediate view. In addition, they can identify and sketch the implications resulting from the processing of an entity, which is facilitated by the `getImplications` method. Each `Processor` is tailored to interact with a particular entity type, enhancing the framework's ability to handle data structures.

The specialization of processors is further emphasized by their ability to identify and declare the specific entity type they are equipped to process, an attribute accessible through the `getProcessableType` method. This specialization supports that each `Processor` operates within its defined domain of expertise, contributing to the efficiency and soundness of the model-driven composition and refinement framework in handling model-driven operations.

The framework introduces two abstract processors to address the needs of model-driven composition and refinement. The first, named `Processor`, encapsulates the implication, composition, and refinement mechanisms applicable to a wide range of wrapped elements or relationships. The second, the `RelationProcessor`, builds on these basic functionalities to specifically address generic relationships' processing. This `Processor` incorporates logic for replacing placeholders, whether direct or indirect, thereby streamlining the refinement process and supporting the integrity and coherence of the resulting view.

5.4.4. Orchestrator

Within the model-driven composition and refinement framework, an orchestration component named the `Orchestrator`, plays a critical role in facilitating the processing of entities. This `Orchestrator` is designed to house the processors and the intermediate view integral to the framework's operations. It supports that the intermediate view is easily accessible to internal processes via the `getView` method. The `Orchestrator` is tasked with managing

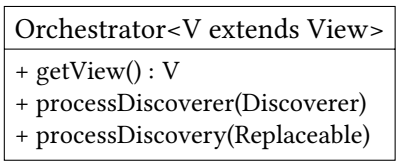


Figure 5.7.: The `Orchestrator` class orchestrates the processing of entities within the framework, coordinating between different processors and the intermediate view to support data handling. This figure is adapted from the following authored publication: [Gst+24]

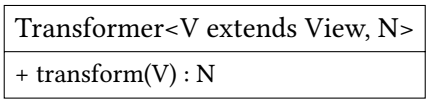


Figure 5.8.: The `Transformer` interface is designed to transform intermediate views into output views of arbitrary types, tailored to meet the needs of view consumers. This figure is adapted from the following authored publication: [Gst+24]

the workflow of entity processing, delegating tasks such as processing discovered entities and their resulting implications, and handling discoverers. This delegation is accomplished through the `processDiscovery` and `processDiscoverer` methods, which assign these tasks to the appropriate processors. This approach emphasizes the role of the `Orchestrator` in seamlessly coordinating between the various components of the model-driven composition and refinement framework to support an orderly processing within the model-driven environment.

5.4.5. Transformer

In the model-driven composition and refinement framework, the transformation process is encapsulated by a particular interface called `Transformer`. This interface is designed to take an intermediate view as input and, through its `transform` method, transform that view into an object of a specified but arbitrary type. This ability to transform intermediate views into different object types highlights the applicability of the `Transformer` and its role in tailoring the output to meet various requirements. The framework advocates using multiple instances of the `Transformer` interface to accommodate generating output views with different structures and formats. This approach allows for an adaptable transformation process, enabling the model-driven composition and refinement framework to produce output views tailored to different consumers' specific needs.

6. Quality Prediction Model Integration

This chapter is dedicated to the technical implementation of the Retriever approach, conceptualized in abstract and formal terms. Building on the theories discussed, we now present a concrete, practical application of these principles through the development of four components: the reverse engineering pipeline, model-based view extraction, composition and refinement, and final visualization and validation. Each element is essential to applying our reverse engineering approach to real-world software systems and is described in the following subsections.

Figure 6.1 shows the schematic of our Retriever approach, which shows three steps, each tasked with processing different types of information. The data flow through these steps is represented by directional arrows, indicating the sequential processing and transformation of data. The input and output data are visually annotated with UML notes. These are labeled to correlate directly with the pseudocode detailed in Chapters 4 and 5 and Section 4.1. Each of these steps is encapsulated within rounded rectangles, symbolizing their role as discrete but interrelated steps within the overall process. In addition, a larger, rounded rectangle encapsulates the deployment step, emphasizing its execution as a Continuous Integration (CI) and Continuous Delivery (CD) action that is integral to implementing the approach in real-world scenarios. The following four subsections (Sections 6.1, 6.2, 6.3 and 6.4) explain these steps' interactions, understanding each step's contribution to the reverse engineering framework.

Section 6.1 focuses on the reverse engineering pipeline. This section introduces our open-source retriever approach tailored for automated reverse engineering of software

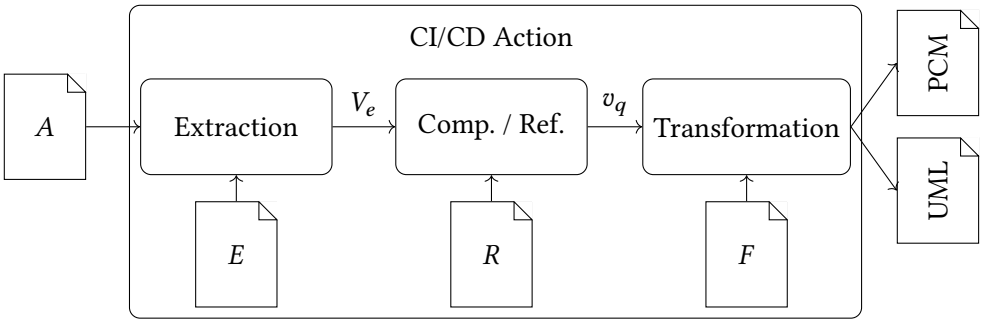


Figure 6.1.: The schematic of our Retriever approach has three steps, each processing different information. The arrows represent the data flow, and the input and output data are represented as UML notes. The rounded rectangles represent the three steps of our reverse engineering approach, and the large rounded rectangle represents deployment as a Continuous Integration and Continuous Delivery (CI/CD) action. This figure is adapted from the following authored publication: [Kir+24a; Kir+24b]

architecture models. We use the EMF to support integration with popular development environments and input and output options that improve applicability. A key feature of our pipeline is the integration of continuous integration and continuous deployment CI/CD practices via GitHub Actions [Git24], which facilitates ongoing model generation and supports architectural verification and documentation processes. This approach emphasizes our tools' applicability in typical software development workflows.

Next, Section 6.2 describes our framework for automating the extraction of software architecture models. Our Retriever approach emphasizes artifact discovery using a suite of technology-specific finders capable of analyzing various software systems. Model-to-model transformations are at the core of our extraction process, implemented using Xtend [Bet16; Ecl24d]. This framework supports multiple technologies, including Docker, ECMAScript, and Java, and allows for adding new extraction rules while maintaining adaptability to meet evolving project requirements.

We examine composition and refinement in Section 6.3. We are implementing our model-driven composition and refinement processes using the PCM. Written in Java, this framework illustrates the adaptability of our approach and its relevance to real-world software architecture analysis. The choice of the PCM helps provide a standardized representation of architectural components for uniform performance analysis across different projects.

Finally, Section 6.4 outlines our strategies for transforming intermediate representations into final, architecturally meaningful outputs. Our approach uses both the PCM for in-depth quality with architectural analysis and PlantUML [Roq+24; Hal+24] to generate understandable UML diagrams. Our adoption of technology-agnostic finalization rules supports various architectural description languages, enhancing the applicability of our framework.

These subsections detail the practical applications of our theoretical concepts and illustrate our approach's applicability in tackling the complexities of reverse engineering in software environments. This chapter aims to provide the reader with a thorough understanding of the technical underpinnings that facilitate practical reverse engineering of software architectures, thereby bridging the gap between abstract theoretical formulations and their practical implementations.

> The subsequent sections are based on the following authored publications: [Kir+23a; Kir+23b; Gst+24; Kir+24a; Kir+24b]

6.1. Reverse Engineering Pipeline

Our novel Retriever approach to automated reverse engineering of software architecture models has been released as an open-source project on GitHub [Kir+24d; Kir+24c] under the Eclipse Public License 2.0 (EPL-2.0). This decision is intended to increase the accessibility and collaborative development of our approach by inviting contributions from a wide range

of researchers, developers, and practitioners. The software is implemented using the EMF, which supports model-driven engineering capabilities. It is accessible as a plug-in for the Eclipse IDE, facilitating seamless integration into existing development environments.

The technical architecture of our Retriever approach is designed to incorporate third-party libraries critical for processing inputs, thereby increasing the system's quality. We use the PCM metamodel to perform an analysis of system architectures. Our approach is designed to provide output options, including both PCM and UML models. This output capability supports the idea that our tool is adaptable to various use cases and user preferences, meeting a wide range of architectural visualization and analysis needs.

In addition, the software is designed to support both the Graphical User Interface (GUI) and Command-Line Interface (CLI), making it suitable for different levels of user expertise. To further integrate with modern software development practices, we have developed GitHub Actions [Git24] as part of our CLI. This feature allows our tool to be easily integrated into CI/CD pipelines. Such integration facilitates the automatic generation of both PCM and PlantUML models directly from the repositories on every code commit, illustrating a practical application of continuous architecture verification and documentation.

The adoption of GitHub Actions emphasizes the practical applicability of our approach in real-world scenarios. It can be configured to continuously update and refine architecture models with each new code commit. This capability promotes better management and tracking of architectural changes over time and reduces the overhead associated with manually updating and maintaining software architecture documentation.

Integrating continuous architectural review and documentation into CI/CD pipelines improves our software development approach. This integration automatically generates quality prediction models directly from software repositories with each code commit. Such automation facilitates the immediate production of updated software architecture models at each code iteration and enables developers and architects to address potential quality issues as they emerge proactively.

This process is essential in maintaining an up-to-date representation of the software architecture, allowing for real-time revisions and improvements aligned with the latest changes in the codebase. Each code transfer initiates an update, supporting the idea that the software architecture models are current and version-controlled. This capability is essential for maintaining historical accuracy and accountability and providing a transparent audit trail of architectural decisions and their evolution over time.

This integration facilitates the ongoing monitoring and assessment of the software architecture's health and stability. By integrating this process into the CI/CD pipeline, the approach supports the idea that architectural assessments are not isolated events, but rather continuous actions that contribute to the overall quality of the software system. It provides developers and architects with the requisite tools and insights to anticipate and mitigate the risks of architectural drift and degradation.

By leveraging one of the most widely used CI/CD technologies, our Retriever approach simplifies integration with existing software projects and improves the collaborative poten-

tial of using Git for version control. This strategy supports the tracking and documenting of all changes and improvements to the architectural models, providing a transparent and accessible means to manage the evolution of the software architecture. Thus, our approach supports the analysis and visualization of software architectures and aligns with modern software development practices, contributing to model-driven development and automated software engineering.

6.2. Model-Based View Extraction

Our framework for automated reverse engineering of software architecture models uses a set of libraries to facilitate the initial discovery of artifacts from different technologies. This discovery step is necessary because it lays the foundation for the subsequent model-to-model transformations that characterize our approach. In particular, our implementation features a set of finders, each tailored to specific programming languages and configuration files, which improves our artifact reverse engineering capabilities. This approach facilitates in-depth analysis of software architectures but also aligns with modern software development practices.

The overarching architecture of our framework is instantiated on top of the PCM, focusing on both architectural and software quality elements. This instantiation is implemented in Java and includes a set of high-level refinement rules that address structural and quality aspects of software engineering. By aligning with the PCM, our framework supports a complete analysis of software quality and architecture, providing an understanding of the software systems it addresses.

The following list provides information on the third-party software components utilized in the core. In addition to the name of the software component, the respective software license and a description of the component must also be provided. Where feasible, the licenses are specified in the standardized short description of the Software Package Data Exchange (SPDX) specification [SPDX]:

Apache Commons CSV (Apache-2.0) An API for parsing and manipulating CSV files. It supports various configurations, including custom delimiters and handling file structures, making it a tool for data exchange and integration tasks in software systems [Apa24a].

Eclipse IDE (EPL-2.0) An open-source IDE that supports software development in various programming languages, including Java, C++, and Python. It offers capabilities for code editing, debugging, testing, integration with version control systems, and build tools [Ecl24a].

Eclipse Java Development Tools (JDT) (EPL-2.0) A set of instruments for Java development within the Eclipse IDE. It includes features for editing, compiling, debugging, and testing Java code and helps developers build Java applications through an IDE [Ecl24b].

EMF (EPL-2.0) A modeling framework and code generation facility for the construction of tools and other applications based on a structured data model. EMF offers the capability to generate Java classes from models described in XML or other modeling languages, thus facilitating the development of data-driven applications [Ecl24c].

Java Properties (GPL-2.0-with-classpath-exception) A persistent set of properties that can be loaded from or saved to a stream. This utility is essential for managing configuration settings in Java applications, allowing developers to externalize configuration data and streamline application deployment and maintenance processes [Ora24].

JDOM (JDOM Project License) An API for reading, manipulating, and writing XML data from Java code. It simplifies interaction with XML documents, making it easier for developers to implement XML data processing and integration tasks within Java applications [HM+24].

JSON in Java (Public Domain) A lightweight library for parsing and generating JSON data, also known as *org.json*. It provides an API for converting between Java objects and JSON, simplifying the processing of JSON data within Java applications and enhancing the ability to handle web-based data exchange formats [Lea+24].

JSqlParser (Apache-2.0) A library that parses SQL statements into a hierarchy of Java classes. It supports various SQL dialects, allowing developers to programmatically analyze, modify, and generate SQL queries to build database management tools and perform SQL-based operations within Java applications [JSQ24].

Maven (Apache-2.0) A software tool that is employed for the build automation of Java projects. The software tool delineates the processes and dependencies for software build, presents a model for project management, and facilitates the centralized storage and dissemination of information necessary to the build, reporting, and documentation of projects [Apa24b].

PCM (EPL-2.0) A framework for modeling and analyzing the quality attributes of component-based software architectures. It enables architects to predict the impact of design decisions on system quality attributes, providing valuable insights for optimizing the software architecture [Reu+16].

PlantUML (EPL-1.0) A tool that allows developers to create UML diagrams from plain text descriptions. It integrates with various development environments, including Eclipse, to facilitate the visualization of software architectures and design patterns, improving the documentation and communication of software designs [Roq+24; Hal+24].

Project Nashorn (GPL-2.0-with-classpath-exception) An ECMAScript engine developed in the OpenJDK project that enables JavaScript and TypeScript code to run on the Java Virtual Machine (JVM). It makes integrating ECMAScript code into Java applications easy, providing a seamless bridge between the two languages and improving interoperability between software components [Ora21].

SnakeYAML (Apache-2.0) A parser and emitter for Java that supports parsing YAML documents into Java objects and serializing Java objects into YAML documents. This tool is essential for applications that rely on YAML for configuration management, data serialization, and inter-service communication [Aso+24].

Snyk CLI (Proprietary) A CLI for Snyk's security tool that helps developers find and fix vulnerabilities in their code, dependencies, containers, and infrastructure configurations. This tool integrates with various development workflows to improve security posture throughout the software development lifecycle [Sny24].

SoMoX (EPL-2.0) A tool for extracting software behavioral models from existing code bases. It performs static analysis to transform Java AST into Service Effect Specification (SEFF) in PCM representations, facilitating reverse engineering of software architectures for performance analysis [Kro+24].

Xtend (EPL-2.0) A dialect of Java designed for use within the Eclipse IDE. The language compiles into Java code, integrates seamlessly with existing Java libraries, and provides additional productivity features such as type inference, extension methods, and lambda expressions [Ecl24d].

6.2.1. Artifact Parser Libraries

Adopting libraries for model-based artifact extraction is generally preferred over the labor-intensive process of custom library creation. This preference is driven by several persuasive factors, including improving applicability in developing software architecture models. One advantage of leveraging established libraries is the pre-existing implementation of extraction functionalities. This out-of-the-box capability saves development time and resources and bypasses the complexities associated with the ground-up development of such functionalities.

These libraries are typically under continuous maintenance, which includes regular updates and bug fixes. Such diligent upkeep helps the library remain compatible with ongoing advancements and changes in programming languages and technology standards. For instance, the Java programming language frequently undergoes updates that could necessitate adjustments in the artifact extraction interface to maintain compatibility with new language features.

The user base associated with popular libraries encourages an active community forum or other support mechanisms. These platforms serve as valuable resources for troubleshooting, sharing insights on everyday issues, and disseminating updates; thus, well-established libraries usually reflect adherence to best practices in model-based artifact extraction.

The collective wisdom encapsulated in these mature libraries, distilled from years of development and refinement by experienced developers, offers an asset to users. This shared knowledge base assists users in leveraging mature strategies and avoiding common pitfalls in software architecture model development. These libraries are generally designed for seamless integration with other tools and frameworks within the user's programming

ecosystem. Such integration capabilities streamline the development workflow, enhance productivity, and reduce potential friction points.

For example, our framework's Java Source Code Finder is based on the JDT [Ecl24b], which supports up to Java 21 archives and properties files. This enables the parsing and analysis of Java code, which is essential for extracting architectural elements from Java-based applications. Similarly, we use the OpenJDK Nashorn engine [Ora21] to discover ECMAScript 6 scripts, and JSqlParser [JSQ24] to identify SQL statements in the codebase. Our approach also includes specialized finders for JSON, XML, and YAML configuration files, leveraging third-party libraries to handle these standard data formats.

The use of widely recognized libraries also aids in maintaining compatibility across various components of a software project. It minimizes compatibility issues when interacting with different segments of a codebase or collaborating with other developers. This approach simplifies the development process and improves the software development lifecycle, reinforcing the preference for established libraries over developing proprietary solutions.

6.2.2. Extraction Rules Implementation

Once artifacts are identified and collected, they are processed using our model-to-model transformation rules implemented in Xtend. This programming language supports imperative and functional paradigms [Ecl24d]. Xtend, an extension of Java, provides a platform that facilitates declarative and imperative transformation definitions. This language choice not only simplifies the definition of transformation rules, but also improves the expressiveness and maintainability of our code.

Our approach currently implements extraction rules for a curated set of technologies, including Docker [Doc24], ECMAScript [ECMA], Gradle [Gra24], JAX-RS [JAXRS], Maven [Apa24b], and Spring [VMw24]. The extraction rules for these technologies have been developed based on the available documentation, ensuring independence from the specific software systems that use them. These technologies were chosen because of their widespread use in component-based REST web services and their importance in conveying architectural information critical to our analysis. The selection of these technologies highlights our approach's ability to handle modern software development environments and its adaptability to different architectural styles.

Our approach is designed with extensibility in mind. It allows the addition of new extraction rules for supported or emerging technologies at runtime, thus meeting the evolving needs of software projects. This feature is particularly relevant in a project-specific context, where developers or project experts may need to customize the mapping of artifacts to architectural elements. Such customization can be achieved by integrating additional, user-defined extraction rules into the CI/CD pipeline, enhancing the approach's responsiveness to specific project requirements.

6.2.3. Static Vulnerability Analysis

To improve software security, static analysis is a central technique for examining development artifacts, including source code and configuration files. This analysis is performed using third-party software tools that automate the detection of vulnerabilities in software products. These include static dependency analysis tools, such as Snyk [Sny24], which examine software dependencies and compare them against databases of known vulnerabilities to identify potential security risks.

While these tools identify existing vulnerabilities, they typically do not uncover new, previously unidentified vulnerabilities. Their functionality is limited to known issues in code, open-source dependencies, or container images. Specifically, Snyk uses a CLI for vulnerability discovery. This Snyk CLI, integrated into the security view extraction step, performs analysis to locate vulnerabilities within the project, supporting various environments, including Java projects using Gradle and Maven build systems.

The output of such analysis includes CVE numbers, each associated with specific development artifacts. For example, the Snyk CLI extends its scan to include a project's build configurations, such as Gradle's `build.gradle` or Maven's `pom.xml` files. This capability allows the CLI to be integrated locally or within a build pipeline, making it easier to check open-source dependencies for known vulnerabilities. However, it is essential to note that Snyk's current capabilities are limited to identifying statically known and publicly documented vulnerabilities. This limitation excludes potential unknown attack vectors from detection.

In addition, the vulnerabilities identified by Snyk are correlated to architectural elements based on code or dependencies. To reconstruct architectural and security models, future improvements will extend this correlation to additional artifacts, such as containers and infrastructure-as-code elements. This approach highlights the potential for integrating other analysis tools that can associate CVEs with code or configuration files, expanding the scope of vulnerability detection.

Snyk's process involves comparing file signatures against a database of known vulnerable files. Suppose a vulnerability is addressed in code in ways apart from updating dependencies, such as direct changes to the code base. In that case, these changes are not always detected, which can lead to an overestimation of exploitable vulnerabilities. In addition, discrepancies can arise when dependencies are specified in a code file but not actively used in that file, leading to a potential overestimation of risk.

Our approach is theoretically designed to accommodate any analysis tool that can associate CVEs with code or configuration files, thereby increasing the applicability of the security insights generated. This capability streamlines the security analysis process and supports an accurate match between identified vulnerabilities and their corresponding architectural components.

6.3. Composition and Refinement

For the model-driven composition and refinement step, a metamodel-dependent framework has been concretely instantiated for the PCM, demonstrating the applicability of the model-driven composition and refinement framework in real-world applications. This PCM-based instance, developed in Java, is publicly available on GitHub [Kir+24d; Kir+24c] under the EPL-2.0, encouraging open collaboration and use in the software architecture community. PCM was chosen for this instantiation because of its support for architecture-based quality prediction. PCM facilitates automated optimization regarding quality properties and supports graphical visualization of these properties, thereby enhancing the interpretability and analysis of software architecture designs.

The implementation serves a dual purpose: first, as a practical application to validate the composition and refinement step, and second, to examine the feasibility of using high-level programming languages, specifically Java, to define refinement rules within the framework. Using the PCM metamodel is strategic; it provides a standardized representation of software architectures for supporting compatibility and accuracy in view processing. This standardization is critical for capturing architectural constructs such as components, connectors, and interfaces, which are essential for the intermediate representation in this step.

While the current implementation focuses on PCM, the underlying concepts of model-driven composition and refinement are designed to be adaptable to other ADLs. This adaptability emphasizes the framework's applicability and potential for extension beyond PCM-based systems. The model-to-model transformation rules, which are essential for composing and refining views, are implemented in Java. Java's effectiveness influences this choice of imperative and object-oriented programming paradigms, which are advantageous for structuring transformation rules using object-oriented features such as inheritance.

Java's compatibility with the existing components of the model-driven composition and refinement framework supports seamless integration and compatibility throughout the system, which simplifies maintenance and increases codebase coherence. In addition, the architecture of the framework supports the integration of other transformation languages such as Xtend [Bet16; Ecl24d], ATL [JK06; Ecl23], or QVT [Kur08; QVT], providing further adaptability in the choice of tools for implementing refinement rules.

The refinement rules are designed to apply to component-based systems, suggesting utility across different technology domains. They are also designed to be combinable with any extraction rules and technology-agnostic, underscoring the framework's ability to serve different system architectures and its potential for reuse in different component-based system configurations.

The PCM is continuously evolving and expanding, which requires a selective implementation of its metamodel elements and relations within the model-driven composition and refinement framework. To this end, a focused subset of six element wrappers and nine relation wrappers has been developed that encompass components of the PCM to support a model representation. The elements selected for implementation include *Operation*

Signature, *Operation Interface*, *Basic Component*, *Resource Container*, *Communication Link Resource Specification (CLRS)*, and *SEFF* each chosen for their relevance to architectural and performance modeling within software systems.

Implementing these wrappers allows for the encapsulation of PCM elements, facilitating the modeling of structural and behavioral aspects of software architectures. In particular, the SEFF and CLRS elements are critical for integrating performance-related information into architectural models. The SEFF wrapper models the performance characteristics and behavior of services provided by components, while the CLRS wrapper details the performance characteristics of connections between resource containers. This dual focus on architectural structure and performance metrics improves the framework's applicability and enables it to support quality predictions.

This step's ability to generate PCM models enriched with performance and architectural data emphasizes its applicability in leveraging model-driven reverse engineering techniques for practical applications. The generated models reflect the system's current state by incorporating architectural and performance information. As a result, the framework serves as a tool for software architects and engineers to improve software quality through accurate, model-driven analysis.

A concrete instantiation of the framework components is undertaken to implement the PCM-based instance of the framework. This involves implementing the generic capabilities of the framework to concretely address the requirements of PCM, reflecting the need for extracting views from artifacts. A generic discoverer tailored for evaluation purposes is added to the framework, setting the stage for concrete instantiation.

A part of this adaptation is the instantiation of the surrogate component, which includes integrating a concrete PCM intermediate view. This specialized view facilitates the handling and manipulation of PCM-specific data within the framework. In addition, specific wrapper classes are developed for each element and relationship within the PCM, ensuring that each component and its relationships are accurately represented within the system.

The processor component of the framework was also instantiated concretely for the PCM. The abstract and relation processors are implemented for each element and relation type in the PCM. This allows precise processing tailored to the structural and relational specifics of the PCM architecture. In addition, to improve the functionality of these processors, PCM-specific refinement rules are translated into Java and integrated into the system. These rules support the correct interpretation and modification of model elements according to PCM standards.

The transformation component of the framework is also concretely instantiated for the PCM to include four different transformer realizations. Each transformer corresponds to one of the four view types used in PCM: repository, system, assignment, and resource environment. This tailored approach allows this step to generate specific views of the PCM essential for architectural analysis and quality assessment.

Eight PCM-specific refinement rules accompany these technical improvements, formulated in natural language and then translated into Java to facilitate the precise refinement of

Identifier	Rule expressed in natural language
r_1	At least one container is associated with one component.
r_2	An interface is provided by one or more components.
r_3	A signature is provided by one or more interfaces.
r_4	A component has one service effect specification for each signature it provides.
r_5	A container link is provided by exactly one link specification.
r_6	A link specification is associated with at least one container link.
r_7	An assembly of components requires all associated containers to be linked.
r_8	A container link may not be replaced by indirect placeholders.

Table 6.1.: The eight implemented refinement rules of the PCM-based instance of the model-driven composition and rule-based refinement framework, formulated in natural language

information within the framework. These eight natural language rules are listed in Table 6.1 with an identifier. These rules include supporting that each component is associated with at least one container, each interface is provided by at least one component, and one or more interfaces provide signatures. Each component has a service effect specification for each signature it provides. In addition, container links must be specified by exactly one link specification, a link specification must be associated with at least one container link, and an assembly of components requires that all associated containers be linked. In addition, indirect placeholders may not replace a container link, underscoring the framework's commitment to maintaining integrity and accuracy in modeling relationships.

Together, these improvements strengthen the framework's ability to handle PCM-based architectural models, enabling automated reverse engineering and quality prediction in software architecture models. Through these extensions and rule formulations, the framework is better equipped to handle aspects of software architecture modeling, thereby contributing to the field of software engineering.

6.4. Final Views and Validation

Our approach for automated reverse engineering of software architecture models utilizes a transformation process that uses both the PCM and PlantUML [Roq+24; Hal+24] to generate outputs. These outputs are tailored to facilitate in-depth software architecture and quality analysis, supporting various stakeholder needs.

The PCM output captures aspects of software quality and architecture. This specificity enables users to perform model-based quality analysis, providing insight into the software system's architecture. The structured nature of PCM facilitates a breakdown of system components, their interactions, and dependencies, which is critical for assessing system quality and planning maintenance strategies.

Conversely, the PlantUML output serves a purpose by visually representing the software system's architecture through universally recognized UML diagrams. These diagrams provide an understanding of the system's structure, making it accessible not only to technical stakeholders but also to those who may not be familiar with the technical details, such as project managers and non-technical decision-makers.

To generate these outputs, our Retriever approach incorporates finalization rules that guide the transformation of intermediate views into final views that conform to both PCM and UML metamodels. These rules are designed to be generic enough to be applied across different ADLs. Implemented in component-based systems, these finalization rules support the idea that the output remains coherent and accurate regardless of the underlying technology or specific architectural style.

Applying these rules is technology-agnostic and synergistic with any set of extraction and refinement rules previously used in the process. This integration allows for a seamless transition from raw data extraction to final view generation, supporting all transformations that align with the reverse engineering process's goals.

Establishing such rules improves the applicability of the reverse engineering process across different modeling environments while supporting the transition from model integration to final representation. This supports the idea that the final software architecture models reflect the underlying system architecture and are compatible with the analytical and developmental tools that operate within various ADL frameworks.

Developing additional finalization rules is important for accommodating software ADLs, including many UML tools. These rules must be adapted to align with the specific metamodels of the respective ADLs, supporting seamless integration within our Retriever approach. This adaptation process entails mapping ADL concepts to the derived constructs and outcomes from the preceding composition and refinement steps. It is important to establish precise rules that direct the transformation of intermediate architectural views into finalized ADL-specific views.

The distinction between finalization and extraction rules is particularly relevant to our Retriever approach. In contrast to extraction rules, which interact directly with and are dependent upon software artifacts, finalization rules operate independently of these artifacts. Conversely, they are based on the outcomes of the preceding composition and refinement steps. The idea of finalization rules lies in their ability to transform the aggregated views—those that have been previously extracted, refined, and composed—into coherent and structured software architecture models. The models are tailored to meet the requirements and specifications of the chosen ADLs.

Our framework uses Palladio's multi-view approach to represent different facets of the system architecture through three final views: structure, behavior, and deployment. Each of these views serves different purposes and addresses different aspects of system analysis:

Structure View Focuses on the static aspects of the system, such as components, connectors, and configurations.

Behavior View Addresses the dynamic aspects, detailing how components interact over time and under different conditions.

Deployment View Provides insight into the operational environment, detailing how components are distributed across hardware resources and their runtime dependencies.

Attack View Provides a view of potential vulnerabilities and the requirements to exploit them, considering both permission levels and attacker context.

This separation of concerns is helpful for managing the complexity of modern software systems and facilitating targeted analysis tailored to specific system aspects. For example, while the deployment view may focus on runtime configuration and resource utilization, the structure view can be used independently to analyze the system's high-level design.

By leveraging the capabilities of both PCM and PlantUML in our finalization process and adhering to Palladio's structured approach to system views, our framework supports reverse engineering of complex software architectures. This capability is helpful for maintaining the relevance of software systems in an ever-evolving technology landscape.

Part III.

Evaluation and Discussion

7. Experimental Evaluation

This chapter presents an experimental evaluation of the Retriever approach to automated reverse engineering of software architecture models. It evaluates the practical applicability of the Retriever approach in real-world use cases, demonstrating its potential to improve system understanding and facilitate architectural decision-making. The evaluation shows that the Retriever approach can meet the criteria for generating functional and valid architectural models and illustrates its accuracy in different technological and architectural contexts.

This chapter illustrates the value of key use cases that link theoretical advances with practical applications, supporting the approach's accuracy in real-world environments. In addition, the approach and associated case studies are available on GitHub [Kir+24d; Kir+24c], an open-source platform, to increase transparency and facilitate peer replication and evaluation. Our approach's integration into CI/CD pipelines, applicability to different technology environments, practical scalability to different project sizes, and comparison with gold standards serve to validate it.

Our evaluation uses the GQM approach, which provides a systematic structure to the research process. This approach requires the definition of precise objectives, formulating probing questions, and linking quantifiable metrics to these questions, thereby enabling the collection of empirical data. A curated selection of popular systems on GitHub supports implementing relevant and reproducible evaluation processes. The evaluation analyzes the open-source systems to develop specific extraction rules and presents the results in a structured format to demonstrate our approach's applicability and accuracy.

The evaluation shows that our approach can practically produce accurate architectural abstractions. However, some manual intervention is still required, suggesting potential avenues for future extension and automation. The chapter concludes by examining the possible threats to the experimental evaluation's validity based on Runeson's guidelines. Our evaluation includes internal, external, and construct validity and reliability to support the accuracy of our experimental evaluation and the credibility of the findings.

> The subsequent sections are based on the following authored publications: [Kir+23a; Kir+23b; Kir+24a; Kir+24b]

7.1. Key Use Cases

In general, key use cases represent an aspect of the empirical validation process in scientific research. These use cases bridge the gap between theoretical development and practical implementation, supporting that new advances are applicable when deployed in their intended environments. In software engineering, key use cases are frequently employed to evaluate the efficacy of development tools, algorithms, or architectures in processing real-world data and operations.

To support transparency and replicability, our approach and associated case studies are open-source and accessible on GitHub [Kir+24c], allowing peers to replicate and evaluate our validation procedures. In addition, the experimental evaluation of our approach is entirely available on GitHub [Kir+24d], showcasing our commitment to open science and the sharing of knowledge within the research community. The open-source nature of our approach supports the idea that it is not only a theoretical framework but also a practical approach that can be readily examined, used, and modified by other researchers and practitioners. We demonstrate the approach's maturity and simplicity by making our approach, case studies, and experimental evaluation openly available. This transparency facilitates the validation and replication of our results, allowing others to verify and build upon our work without barriers.

The following four key use cases will be used to evaluate the applicability and accuracy of our experimental development. These four use cases represent use cases in which our approach could be used and describe the requirements for the planned use cases.

Integration into CI/CD Pipelines CI/CD pipelines are a cornerstone of modern software development practices and, as such, are an aspect of this validation. The objective is to ascertain whether our approach can automatically update architectural models with each code commit or during deployment phases. This will evaluate not only the compatibility of our approach with current software development paradigms but also its responsiveness to code changes. This will determine its potential as an agile development tool.

Applicability to Diverse Technology Environments The approach will be applied to software systems built on different technology stacks to determine their applicability and breadth. Each of the stacks mentioned above – including Java to ECMAScript, Maven to Gradle, or YAML to XML – presents different data extraction and architectural modeling challenges. This use case is designed to highlight the ability of our approach to accurately delineate and model the distinctive architectural characteristics of each technology, thereby demonstrating its applicability.

Scalability Across a Range of Project Sizes Our approach is applied to various projects, including small applications with few components and large enterprise systems with many modules to evaluate practical scalability. This use case is essential to evaluating the performance scalability of the approach and providing insight into its practicality in various real-world settings. This allows for determining its applicability on a larger scale.

Comparison with Manual Modeling The models generated by our approach will be compared to those manually constructed by experienced architects for identical software systems. This comparison is designed to validate the accuracy of the results produced by our approach. By comparing automated and expert-generated models, we can measure our approach's accuracy in capturing architectural elements and their interrelationships.

These key use cases challenge our approach from multiple views, including its operational integration and ability to handle different technology environments and scales. Together, they evaluate the applicability and accuracy of our approach to automating the reverse engineering of software architecture models. This evaluation contributes to the theoretical and practical understanding of architectural model generation from various inductive software views.

7.1.1. Systematic Goals

In our experimental evaluation, we employed the GQM approach, as formulated by Basili [BW84; Bas92; BCR94], to conduct a rigorous evaluation of our Retriever approach. The structured approach comprises three distinct steps that facilitate a systematic investigation of the approach.

Goals (G) First, specific goals were defined for the Retriever approach, focusing on various aspects, including effectiveness in different environments. The quality of the generated architectural models and the approach's applicability to different technology ecosystems are also considered. Several factors are considered in this phase, including the intended purpose, target audience, and operational context. This enables the evaluation to be tailored to practical implementation scenarios.

Questions (Q) Subsequently, a series of probing questions is developed to elucidate the intricacies of the Retriever approach. The questions are designed to ascertain the extent of the model's capabilities and to determine the effectiveness of the Retriever approach in meeting its defined objectives. The questions are designed to evaluate both the functional aspects of the model, including its accuracy and completeness, and the non-functional aspects, such as its usability and scalability.

Metrics (M) Finally, each question is associated with specific metrics that provide quantifiable measures to answer the questions posed. The selected metrics have been chosen to accurately reflect the essence of the questions and provide empirical data that can be analyzed. The metrics may include performance indicators such as the speed of model generation, accuracy rates regarding correct component identification, and user satisfaction.

This GQM approach is essential to ensuring the Retriever approach's evaluation is comprehensive and objective. By establishing clear objectives, formulating pertinent inquiries, and defining precise metrics, we ensure that our evaluation is not only aligned with the theoretical foundations of the approach but also grounded in empirical evidence.

The overarching evaluation question is to determine the overall value and effectiveness of the Retriever approach in practical scenarios. In contrast, research-oriented questions are designed to expand the body of knowledge surrounding automated reverse engineering technologies. This dual focus facilitates both the validation of the approach and the exploration of new avenues for further research and development.

Our GQM plan aligns the validation scenarios with measurable outcomes, ensuring the research is rich and goal-oriented. This plan facilitates the evaluation of the approach, promotes transparency, and enables the research community to examine our experimental evaluation critically.

The systematic evaluation is designed to confirm that the Retriever approach is not merely a theoretical construct, but a rich approach capable of delivering tangible benefits in software architecture. Utilizing a goal-oriented, top-down, and model-based measurement approach, such as the GQM, provides a rich foundation for this evaluation, ensuring the results are valid and actionable.

G1: Demonstrate the approach's applicability to relevant software systems to generate an abstraction of the system as an architectural model.

Q1.1: To what extent are the systems relevant, regarding their use by the community, their size, and their use of heterogeneous technology?

M1.1.1 The relevance of GitHub projects is measured by four key metrics: commits, contributors, forks, and stars.

M1.1.2 For technology usage, two metrics are used: the number of LoC (excluding whitespace and comments) and the number of source files.

M1.1.3 For system views, the chosen metric is the number of different technologies, each providing a unique view of the system.

Q1.2: To what extent is the approach scalable and transferable to relevant real systems?

M1.2.1 The average execution time plus standard deviation of GitHub Actions for comprehensive model extraction.

M1.2.2 The cumulative number of LoC across all artifacts of the software system considered in our approach.

M1.2.3 The number of components in the final model.

M1.2.4 The number of different technologies in a case study.

Q1.3: Does the approach lead to valid instances of an ADL that abstract from the source code?

M1.3.1 When evaluating the validity of architectural models, the primary metric is the number of violated OCL constraints in the reverse-engineered models.

M1.3.2 Regarding abstraction, the secondary metric is the ratio of source files to the number of components in the architectural model.

G2: Evaluate the accuracy of the approach in generating both structural and behavioral properties of a software architecture model.

Q2.1: Have the structural properties of the component-based software architecture been correctly identified compared to the ground truth generated by experts in the field?

M2.1.1 The precision for the components, connectors, and provided and required interfaces.

M2.1.2 The recall for the components, connectors, and provided and required interfaces.

M2.1.3 The F_1 score for the components, connectors, and provided and required interfaces.

Q2.2: Has the behavior (SEFF) of the component-based software architecture with dependency properties been correctly identified, such that data and control flows exist across component boundaries that are not connected at compile time, compared to an expert-generated ground truth?

M2.2.1 The precision for the external calls in the SEFF.

M2.2.2 The recall for the external calls in the SEFF.

M2.2.3 The F_1 score for the external calls in the SEFF.

Q2.3: How accurately can we transfer vulnerabilities from static security analysis to the architectural vulnerability model, compared to expert-generated ground truth?

M2.3.1 The precision for the vulnerabilities in the architectural model.

M2.3.2 The recall for the vulnerabilities in the architectural model.

M2.3.3 The F_1 score for the vulnerabilities in the architectural model.

Q2.4: How much effort is required to define more accurate project-specific rules for an already integrated technology?

M2.4.1 The LoC required to define the new project-specific rules.

M2.4.2 The precision recall, and F_1 score for the components, connectors, and provided and required interfaces.

M2.4.3 The precision, recall, and F_1 score for the external calls in the SEFF.

7.1.2. End-to-End Case Studies

Our experimental evaluation initially focused on gathering and analyzing statistical data to identify the most prevalent and widely utilized technologies for implementing and deploying web services systems. This data informed the development of specific extraction rules tailored to the chosen technologies. These technologies were selected for their prevalence in modern component-based architectures such as web services and microservices and for the architectural information they typically encapsulate. These technologies are selected based on their popularity and ability to solve various architectural problems and provide insights into the system's structure.

The initial phase of our experimental evaluation examines the prevailing technologies utilized in developing and deploying microservices and web services. This phase employs statistical data from industry-recognized platforms, including *Stack Overflow* [Sta23], *fortiss* [for23; KBP20], and *Thoughtworks* [Tho24; Par+24]. The objective is to establish a foundation for developing extraction rules tailored to the most prevalent technologies in space exploration.

The data indicates a preference for Java for back-end development in microservices and web service architectures. Java's prevalence is attributable to the availability of many frameworks that facilitate the implementation of web services. Among these, the Spring Framework is a popular choice for building microservices because its tool suite supports everything from configuration to security.

Regarding architectural design, the REST style is the dominant approach to microservices and web services. This preference is supported by the simplicity and scalability of REST, which lends itself well to the distributed nature of such systems. The JAX-RS specification, which facilitates the development of REST web services using the Java API, has been identified as a tool for implementing REST architectures in Java applications.

Maven and Gradle are regarded as the most commonly used automated build management tools in software development. These tools are critical for managing project dependencies, building processes, and continuous integration and deployment pipelines. They are also crucial for the agile development cycles typical of microservice architectures.

Docker is maturing as a technology in the container virtualization space. Docker's containerization technology enables the uniform deployment of applications across environments, thereby ensuring that microservices remain lightweight and modular. This capability is essential in a microservice architecture, where services are scaled and updated independently.

On the client side, ECMAScript, the standardized core of JavaScript and TypeScript, has established itself as the dominant language for developing web client applications. Its prevalence in front-end development is due to its ecosystem of libraries and frameworks, including React and Angular, which facilitate the development of web interfaces.

The developed extraction rules were applied to a curated selection of systems, selected based on their relevance, popularity on GitHub, and the availability of source code. This

supports our evaluation, is grounded in current software development practices, and improves our findings' applicability. Focusing on popular and widely adopted technologies and frameworks ensures that our reverse engineering approach remains relevant and practical for current and future software development trends. The implementation of the extraction rule for the selected technologies has already been discussed in this section. Consequently, our process for experimental evaluation entails the implementation of extraction rules for the following curated set of technologies:

- Maven [Apa24b] and Gradle [Gra24] for build automation.
- Docker [Doc24] for container virtualization.
- Spring [VMw24] for web application development.
- ECMAScript [ECMA] for web client application implementation.
- JAX-RS [JAXRS] for REST web services implementation.

The selection of software systems for the end-to-end case studies was based on the previously identified technologies. The extraction rules were defined for these technologies so that the systems use the supported technologies, but the extraction rules are independent of the systems. The software systems selected in the studies were chosen to support a representation of real-world applications, thereby enhancing the validity of the evaluation use cases. The chosen systems include popular open-source projects active on GitHub, thus facilitating a transparent and reproducible evaluation process. The evaluation, therefore, tests the applicability of our approach to extracting and reconstructing software architectures from diverse technological bases and application domains.

Our 34 end-to-end case study selection is presented in Table 7.1 in a structured format that includes the respective GitHub identifiers, a brief description, distribution metrics, and technology metrics of each repository. These elements are critical to an understanding and analysis of each system. Each system is presented in tabular form, with the data arranged in ascending order based on its GitHub identifier. This structured presentation allows for easy system comparison and emphasizes our selection process. By presenting systems that vary widely in size, technology, and functionality, we aim to demonstrate the applicability of our reverse engineering approach across a wide range of real-world applications. This approach supports the idea that our findings are grounded in diverse operational contexts, thereby enhancing the generalizability and applicability of our research in automated reverse engineering of software architecture models.

To illustrate this point, Figure 7.1 is provided that compares the three metrics for the size of the case studies utilized. The relative sizes of the technologies are represented by the ratio of the sum of the source files to the sum of the lines of code. The three metrics are presented in Table 7.1 form for convenience. This illustration demonstrates that the end-to-end case studies employed represent a diverse range of software project sizes.

GitHub Identification Each system in our experimental evaluation is identified by its GitHub identifier, which combines the user or organization name and the repository name, separated by a slash. This identifier serves as a locator for a repository

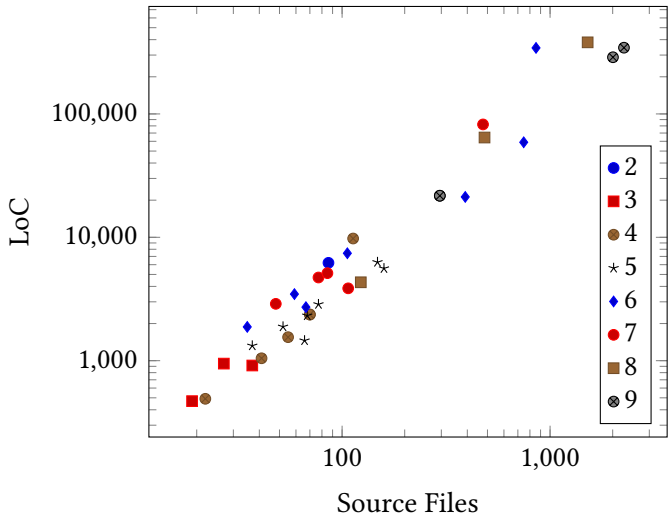


Figure 7.1.: To illustrate the diverse range of sizes and technologies employed in the case studies, the ratio of the sum of source files to their sum LoC and the technologies employed are presented. The numbers in the legend indicate the technologies used in a given case study. The precise numbers are in Table 7.1.

within the GitHub platform and is part of the URL that provides direct access to the repository. To illustrate, the repository `piggymetrics` belonging to the user `sqshq` can be accessed via the URL `https://github.com/sqshq/piggymetrics`.

Repository Description Each system is accompanied by a brief description, usually taken from the documentation provided by the developers. This description gives a brief overview of the application's purpose and functionality.

Distribution Metrics The popularity and activity of each repository are measured using GitHub's metrics, including the number of stars, forks, and commits. The number of stars a repository receives indicates its popularity within the GitHub community. The number of forks suggests the number of times other developers have decided to create a copy of the repository for their development purposes. This indicates the importance of the repository. The number of commits provides a direct measure of the frequency of development activity, displaying the number of updates made.

Technology Metrics Each repository's technology stack is evaluated by analyzing the programming languages and technologies used. The metrics used in this analysis include the number of files and total LoC, categorized by language and technology. This data is collected using the open-source `cloc` tool [Dan+24], which quantifies lines of source code, distinguishing between physical lines, comment lines, and blank lines across programming languages. Such metrics provide insight into the development effort associated with the repository.

`acmeair/acmeair` A Java implementation of the Acme Air sample application that demonstrates a sample microservice architecture.

Distribution		Technology (files / code)	
Commits	77	CSV	1 / 31
Contributors	2	ECMAScript	5 / 708
Forks	156	Gradle	2 / 75
Stars	134	Java	81 / 6207
		Maven	1 / 56
		XML	16 / 349

`anilallewar/microservices-basics-spring-boot` A framework for building complete microservices using Spring Boot and Spring Cloud.

Distribution		Technology (files / code)	
Commits	46	Dockerfile	9 / 126
Contributors	5	ECMAScript	8 / 153
Forks	442	Gradle	9 / 691
Stars	722	Java	37 / 1385
		SQL	1 / 28
		XML	9 / 182
		YAML	34 / 1294

`apssouza22/java-microservice` A complete microservice architecture in Java, including Spring Cloud, ELK for log management, Nginx for load balancing, Docker for infrastructure, and more.

Distribution		Technology (files / code)	
Commits	140	Dockerfile	4 / 97
Contributors	1	Java	116 / 3147
Forks	251	Maven	11 / 1376
Stars	382	XML	9 / 233
		YAML	19 / 743

`callistaenterprise/blog-microservices` Contains sample code for Callista Enterprise microservices blog posts.

Distribution		Technology (files / code)	
Commits	181	Dockerfile	7 / 39
Contributors	1	Gradle	12 / 663
Forks	300	Java	37 / 1401
Stars	407	XML	2 / 31
		YAML	19 / 731

`cloudscale-project/cloudstore` Demonstrates the utility of CloudScale's tools and methodology through a fictional company's e-commerce application.

Distribution		Technology (files / code)	
Commits	200	ECMAScript	2 / 1403
Contributors	4	Java	98 / 7673
Forks	3	Maven	1 / 314
Stars	9	XML	12 / 387

`corona-warn-app` Merge the backend server-side repositories for the Corona Warning App (CWA) into one repository. The CWA was used for contact tracing during the COVID-19 pandemic. It consists of the Corona Warning App Server, the Verification Server, the Portal Server, and the Test Results Server.

Distribution		Technology (files / code)	
Commits	211	Dockerfile	14 / 357
Contributors	35	ECMAScript	14 / 917
Forks	105	Java	1097 / 67 957
Stars	343	JSON	55 / 26 288
		Maven	5 / 617
		SQL	68 / 1479
		XML	57 / 275 666
		YAML	206 / 7189

`corona-warn-app/cwa-dcc-server` Backend implementation for issuing EU Covid-19 digital certificates.

Distribution		Technology (files / code)	
Commits	85	Dockerfile	1 / 8
Contributors	7	ECMAScript	1 / 40
Forks	4	Java	54 / 3130
Stars	7	JSON	1 / 538
		Maven	1 / 120
		XML	1 / 294
		YAML	18 / 593

`corona-warn-app/cwa-log-upload` A log upload service for the Corona-Warn-App that allows developers to analyze log data to identify bugs.

Distribution		Technology (files / code)	
Commits	33	Dockerfile	1 / 12
Contributors	6	ECMAScript	9 / 802
Forks	7	Java	32 / 1644
Stars	4	Maven	1 / 137
		XML	2 / 306
		YAML	14 / 565

`corona-warn-app/cwa-ppa-server` Data donation server for receiving and storing user usage data and event-driven user surveys.

Distribution		Technology (files / code)	
Commits	658	Dockerfile	5 / 155
Contributors	17	Java	294 / 16 237
Forks	12	JSON	5 / 505
Stars	17	SQL	28 / 928
		XML	23 / 1982
		YAML	36 / 1434

`corona-warn-app/cwa-server` Backend implementation for the Apple/Google exposure notification API.

Distribution		Technology (files / code)	
Commits	1406	Dockerfile	5 / 170
Contributors	79	Java	615 / 40 602
Forks	386	JSON	48 / 24 554
Stars	1923	SQL	40 / 551
		XML	31 / 273 343
		YAML	117 / 3638

`corona-warn-app/cwa-testresult-server` Backend implementation for handling test result information from labs.

Distribution		Technology (files / code)	
Commits	110	Dockerfile	1 / 6
Contributors	16	ECMAScript	1 / 39
Forks	49	Java	28 / 1717
Stars	91	JSON	1 / 295
		Maven	1 / 121
		XML	2 / 327
		YAML	14 / 384

`corona-warn-app/cwa-verification-portal` Front-end implementation for the verification process, allowing hotline staff to generate a transaction number for users to upload their diagnostic keys.

Distribution		Technology (files / code)	
Commits	84	Dockerfile	1 / 6
Contributors	22	ECMAScript	4 / 76
Forks	55	Java	20 / 1145
Stars	92	Maven	1 / 97
		XML	1 / 294
		YAML	8 / 260

`corona-warn-app/cwa-verification-server` Backend implementation for the verification process in the Corona-Warn app, validating upload requests from the app and ensuring authorized access to the verification portal.

Distribution		Technology (files / code)	
Commits	211	Dockerfile	1 / 8
Contributors	35	ECMAScript	1 / 40
Forks	105	Java	58 / 3618
Stars	343	JSON	1 / 409
		Maven	1 / 142
		XML	1 / 294
		YAML	22 / 610

`ewolff/microservice` Sample microservice project demonstrating the development and deployment of microservices using Spring Boot and Spring Cloud.

Distribution		Technology (files / code)	
Commits	140	Dockerfile	13 / 43
Contributors	8	Java	42 / 1833
Forks	352	Maven	7 / 380
Stars	723	YAML	8 / 110

`descartesresearch/teastore` The Tea Store system is a microservice reference and test application designed for benchmarking and testing. The system emulates a rudimentary online retail platform for selling tea and tea-related accessories.

Commits	1759	CSV	3 / 360
Contributors	19	Dockerfile	7 / 68
Forks	140	ECMAScript	13 / 1848
Stars	119	Java	200 / 12 304
		JSON	6 / 1303
		Maven	14 / 987
		SQL	2 / 146
		XML	17 / 1816
		YAML	33 / 2892

`fernandoabcampos/spring-netflix-oss-microservices` Sample microservice architecture using Spring Boot and Cloud Netflix library.

Distribution		Technology (files / code)	
Commits	90	Dockerfile	9 / 56
Contributors	1	Java	26 / 560
Forks	14	Maven	9 / 692
Stars	13	YAML	11 / 243

`fudanselab/train-ticket` A train ticket booking system that demonstrates a comprehensive microservice architecture built with Spring Cloud, Docker, and Kubernetes.

Distribution		Technology (files / code)	
Contributors	18	ECMAScript	992 / 197 869
Forks	238	Gradle	2 / 13
Stars	702	Java	635 / 37 738
		JSON	170 / 33 274
		Maven	45 / 2229
		SQL	1 / 13
		XML	1 / 30
		YAML	114 / 16 619

`georgwittberger/apache-spring-boot-microservice-example` An example project that demonstrates a microservice architecture using Apache Camel and Spring Boot.

Distribution		Technology (files / code)	
Commits	7	ECMAScript	1 / 36
Contributors	1	Java	15 / 326
Forks	12	Maven	3 / 108
Stars	9	YAML	3 / 21

`jferrater/tap-and-eat-microservices` A microservices-based restaurant ordering system using Spring Boot, Spring Cloud, and Docker.

Distribution		Technology (files / code)	
Commits	35	Dockerfile	8 / 48
Contributors	1	Java	35 / 624
Forks	6	Maven	9 / 657
Stars	8	XML	1 / 17
		YAML	13 / 111

`kbastani/spring-cloud-event-sourcing-example` A sample project demonstrating event sourcing and CQRS patterns with Spring Cloud.

Distribution		Technology (files / code)	
Commits	35	Dockerfile	12 / 60
Contributors	4	ECMAScript	294 / 75 083
Forks	61	Java	115 / 4741
Stars	78	JSON	3 / 55
		Maven	13 / 1121
		SQL	7 / 117
		YAML	32 / 1024

`kit-recipe-app/recipebackendnew` Backend for a recipe application that implements a microservice architecture for managing recipes and related data.

Distribution		Technology (files / code)	
Commits	83	Dockerfile	1 / 5
Contributors	2	Java	57 / 1977
Forks	0	Maven	1 / 118
Stars	3	XML	2 / 38
		YAML	7 / 183

`kit-sdq/esda` Energy State Data Analysis (ESDA) is a component-oriented reference and test application for performance benchmarking and testing that emulates a message-based energy state data analysis system.

Distribution		Technology (files / code)	
Commits	30	Dockerfile	3 / 12
Contributors	1	Java	7 / 310
Forks	1	Maven	5 / 199
Stars	0	XML	14 / 618
		YAML	8 / 184

`koushikkothagal/spring-boot-microservices-workshop` A comprehensive workshop on building microservices with Spring Boot and Spring Cloud, including examples and labs.

Distribution		Technology (files / code)	
Commits	12	Java	19 / 383
Contributors	1	Maven	4 / 250
Forks	1134	XML	4 / 314
Stars	738		

`mdeket/spring-cloud-movie-recommendation` A microservices-based movie recommendation system using Spring Cloud.

Distribution		Technology (files / code)	
Commits	6	Java	37 / 931
Contributors	1	JSON	3 / 309
Forks	12	Maven	6 / 467
Stars	18	SQL	1 / 51
		XML	5 / 128

`meet-eat/meet-eat-server` Backend server for the Meet-Eat application, a platform for organizing and attending group meals.

Distribution	
Commits	320
Contributors	3
Forks	0
Stars	0

Technology (files / code)	
Gradle	2 / 42
Java	84 / 6169

`openmrs/openmrs-core` The core application framework for OpenMRS, an open-source platform for supporting healthcare in resource-constrained environments.

Distribution	
Commits	1748
Contributors	144
Forks	517
Stars	85

Technology (files / code)	
CSV	2 / 2
Dockerfile	1 / 81
ECMAScript	21 / 4366
Java	1891 / 193 787
JSON	6 / 503
Maven	22 / 4775
SQL	5 / 5464
XML	307 / 134 489
YAML	11 / 787

`openmrs/openmrs-module-webservices.rest` A RESTful web services module for the OpenMRS platform that provides API endpoints for accessing medical record data.

Distribution	
Commits	1748
Contributors	144
Forks	517
Stars	85

Technology (files / code)	
ECMAScript	3 / 79
Java	677 / 54 622
JSON	6 / 503
Maven	15 / 2453
XML	45 / 1201
YAML	1 / 53

`piomin/sample-spring-oauth2-microservices` A sample microservices project that demonstrates the use of Spring Boot and OAuth2 for authentication and authorization.

Distribution	
Commits	6
Contributors	1
Forks	140
Stars	134

Technology (files / code)	
Java	10 / 204
Maven	5 / 167
YAML	4 / 99

`rohitghatol/spring-boot-microservices` Examples and best practices for building microservices with Spring Boot.

Distribution		Technology (files / code)	
Commits	66	ECMAScript	8 / 286
Contributors	2	Gradle	8 / 471
Forks	918	Java	25 / 981
Stars	1769	SQL	1 / 63
		XML	14 / 764
		YAML	11 / 151

`spring-petclinic/spring-petclinic-microservices` A distributed version of the Spring pet clinic application built with Spring Cloud that demonstrates microservice architecture using Spring Boot, Spring Cloud Gateway, and other tools.

Distribution		Technology (files / code)	
Commits	738	Dockerfile	3 / 24
Contributors	35	ECMAScript	21 / 263
Forks	2167	Java	48 / 1167
Stars	1666	JSON	13 / 849
		Maven	8 / 892
		SQL	12 / 207
		XML	2 / 573
		YAML	16 / 345

`shabbirdwd53/springboot-microservice` An example Spring Boot microservice project that demonstrates basic functionality and architectural patterns.

Distribution		Technology (files / code)	
Commits	7	Java	24 / 408
Contributors	1	Maven	6 / 424
Forks	666	YAML	7 / 82
Stars	321		

`sqshq/piggymetrics` A simple, flexible personal finance management tool built with a microservice architecture using Spring Boot, Spring Cloud, and Docker.

Distribution		Technology (files / code)	
Commits	290	Dockerfile	10 / 58
Contributors	13	ECMAScript	8 / 1664
Forks	6112	Java	97 / 3292
Stars	13 224	Maven	10 / 684
		YAML	23 / 588

`webgoat/webgoat` An intentionally insecure application that provides a learning platform for security testing and practicing various attacks and defenses.

Distribution		Technology (files / code)	
Commits	3010	Dockerfile	1 / 31
Contributors	105	ECMAScript	90 / 44 097
Forks	5431	Java	356 / 18 121
Stars	6969	JSON	5 / 181
		Maven	1 / 852
		SQL	16 / 208
		XML	5 / 582
		YAML	10 / 319

`yidongnan/spring-cloud-netflix-example` An example project demonstrating microservice architecture using Spring Cloud and Cloud Netflix library components.

Distribution		Technology (files / code)	
Commits	65	Dockerfile	7 / 52
Contributors	2	Gradle	9 / 142
Forks	367	Java	11 / 292
Stars	807	YAML	14 / 560

Table 7.1.: The selection of our 34 end-to-end case studies is presented in a structured format that includes each repository's GitHub identifier, a brief description, distribution metrics, and technology metrics.

7.2. Applicability Validation

The validation of the applicability of our approach to the automatic reverse engineering of software architecture models is composed of several discrete steps. These include syntactic and semantic correctness, scalability experimental evaluation, and verification of the ability to abstract. Each step is designed to test and scale the applicability of our approach across different dimensions of architectural analysis.

First, the syntactic and semantic correctness of the generated models is verified using tools such as PlantUML and the EMF to ensure that the models adhere to predefined standards and meaningful architectural semantics. This phase confirms the architectural outputs' technical correctness, which is important for their subsequent utility in software architecture analysis.

Subsequently, the approach's scalability is evaluated through the execution of runtime tests in a cloud-based environment, with a particular focus on performance across different system sizes and technology configurations. This analysis provides insight into the approach's applicability in diverse environments.

Finally, we evaluate the capacity of our approach to transform software architectures into analyzable models. This includes examining the correlation between the quantity of source artifacts and the compactness of the resulting architectural models. This analysis demonstrates the efficacy of our approach to simplifying complex code bases into their architectural representations.

Taken together, these steps evaluate our approach's practical applicability. The results confirm our approach's applicability to automatic reverse engineering of software architectures. This structured evaluation serves two purposes. First, it reinforces our approach's theoretical foundations. Second, it demonstrates our approach's pragmatic relevance in real-world scenarios where accurate, scalable, and meaningful architectural abstraction is important.

7.2.1. Experiment Design

The experimentation design for the applicability validation, described in the first goal of our GQM plan in Section 7.1.1, comprises three consecutive steps. The three-step process includes reviewing and evaluating the approach employed through a GitHub Action [Git24] within a CI/CD pipeline. The automatically generated architectural models are subjected to validation and verification for syntactical correctness in the initial phase. This is accomplished using PlantUML for UML diagrams and the EMF framework for PCM models. In the second step, the scalability of the approach is evaluated through the execution of runtime tests in a cloud-based environment. In the third step, the approach's ability to abstract is discussed, focusing on the relationship between source files and components in the models.

7.2.1.1. Syntactic and Semantic Validation

The initial step was to configure a GitHub action within a CI/CD pipeline to automate the execution of our reverse engineering approach directly on GitHub repositories. This action was initiated with each push to the repository, thereby ensuring that the most recent changes were continuously integrated and evaluated. The automated pipeline facilitated the creation and storage of runtime artifacts, streamlining the workflow for continuous architectural analysis.

Subsequently, the generated architectural models were subjected to syntactic and semantic correctness to ensure their accuracy and relevance. This multiphase validation process ensures that the models are structurally sound and semantically meaningful, providing a basis for further analysis and decision-making in software architecture development. This syntactic and semantic correctness phase reinforces our approach's applicability, confirming that the generated architectural models adhere to the technical specifications and agree with the intended architectural semantics.

In this case, some parts of the semantics of the PCM models are defined by the OCL constraints derived from the metamodel. These constraints define the semantics of the PCM models, including the requirement that an assembly context component and the inner role that requires the component must be identical, that a provided role must be bound, and that every user action except start and stop must have a predecessor and successor. For our validation, we consider only these semantics captured by the OCL constraints. Thus, in the following, we refer to the semantics that we can capture with OCL constraints as "semantics".

The conversion of UML models into graphical representations using PlantUML requires the syntactic correctness of the textual input. The textual notation is distinguished by its simplicity, rendering it particularly suitable for transformation. Nevertheless, this process requires the absence of syntactic errors. The process is based on a binary decision mechanism. The textual notation can be readily converted into graphical UML diagrams if it is error-free. However, syntactic errors in the text impede the transformation process and preclude the generation of the corresponding graphical diagrams. The process described ensures that only syntactically correct models are visualized. This process allows for evaluating the syntactic correctness of PlantUML models generated by the GitHub action.

At a more complex level, the PCM metamodels, with their detailed and complete structure, underwent validation using the EMF Validation Framework. This framework has powerful capabilities for evaluating the syntactic and semantic correctness of models developed within EMF. The API enables the definition of specific constraints for each metamodel, thereby enhancing the precision of the validation process. In addition to syntactic checks, semantic checks are performed by analyzing model elements defined in the OCL. The PCM metamodel, an extension of the EMF, supports the definition and validation of OCL constraints, thereby providing a mechanism for enforcing business logic and correctness rules within the models. The OCL constraints were defined independently within the PCM metamodel and subjected to testing for compliance using the EMF Validation Framework. The validation process also used the LALR Parser Generator (LPG) version of the Eclipse OCL environment, which plays an important role in accurately validating OCL constraints. This environment permits the experimental evaluation of logical conditions embedded in the PCM, guaranteeing that the architectural representations meet predefined correctness criteria.

7.2.1.2. Scalability Validation

To evaluate the scalability of our automated reverse engineering approach across diverse software system sizes and technological architectures, we conducted a complete analysis of execution times using GitHub Actions [Git24]. The study was conducted on a cloud-based virtual Linux machine, provisioned explicitly for the project in question, which was integrated into GitHub as a customized runner. This facility enabled precise environment control and customization, a capability provided by GitHub that supports the hosting and customization of execution environments for GitHub Actions.

The test environment consisted of a virtual machine running the Ubuntu 22.04.5 distribution on the Linux kernel 6.5.0. The machine's specifications included an *AMD EPYC 7763* processor with four cores, clocked at approximately 3.5 GHz, and equipped with approximately 16 GB of Random-Access Memory (RAM). The configuration was designed to simulate a typical enterprise-grade server environment closely.

The primary objective of executing the GitHub Action on this configuration was twofold. The first objective was to accurately determine the execution times required for validating the approach under realistic conditions. The second objective was to employ the controlled environment provided by custom GitHub runners to ensure that the timings were constant and not affected by variable external factors.

To ensure the reliability of the results, each case study was executed ten times under identical conditions. The elapsed times, recorded directly within the GitHub Actions workflow, were documented in seconds and rounded to three decimal places for precision. The standard deviation across these ten runs was calculated to evaluate the variability and reliability of the results. This was achieved by utilizing `hyperfine`, an open-source command-line benchmark tool known for accurately performing detailed statistical analysis across multiple executions. The `hyperfine` tool can conduct warm-up runs, clear the cache before each test, and pinpoint statistical outliers, ensuring reliable performance data.

We conducted a comparative analysis with a local test setup to further validate our findings. The local execution was performed on a laptop configured with an *Intel Core i7-8550U* processor with four cores, clocked at approximately 1.80 GHz, equipped with approximately 16 GB of RAM, and running *Microsoft Windows 11 Education* 10.0.22631 as the operating system. This comparison was required to validate the reproducibility and reliability of GitHub Action execution times against a standard local development environment. The comparable execution times between the cloud-based and local configurations validated the reliability of our results and confirmed that our approach exhibits scalability and constant performance across different environments. This serves to illustrate the practical applicability and platform independence of our approach.

7.2.1.3. Abstraction Capability Validation

Our approach is designed to abstract complex software systems into concise architectural models. To evaluate the efficacy of this process, we examine the relationship between the volume of source artifacts in software repositories and the compactness of the resulting architectural models. Specifically, we compare the number of source files and LoC to the number of architectural components in the final PCM instance.

The analysis commences with aggregating the number of source files and LoC for each system under study. This aggregation process entails the summation of the counts presented in Table 7.2, which enumerates these metrics for the various technology stacks

included in each system. The source files and LoC constitute inputs to our reverse engineering process, and their totals reflect the quantity of data processed by our approach to generating architectural models.

The source files for each system include all files relevant to the technologies being analyzed, ensuring an examination of the system's code base. The total LoC, also collected from Table 7.2, represents the content our approach needs to distill into architectural models. As previously described, the `cloc` tool was employed to provide these quantitative metrics, as it offers a reliable measure of the code volume that our approach considers.

On the output side, the number of components identified in the final PCM architecture models is counted. The components are derived from model elements that implement the *RepositoryComponent* interface, qualifying them as architectural components within PCM. This count quantitatively measures the model's complexity and the abstraction process's applicability.

The ratio of LoC and source files to the number of components serves as an indicator of the quality of the abstraction. A lower ratio indicates a higher degree of abstraction, suggesting that our approach condenses the input software system into a simpler model that retains architectural details. This metric thus serves as an evaluation parameter, providing insight into the applicability of our approach to reducing and simplifying complex software architectures into manageable and interpretable models.

This structured evaluation demonstrates our approach's applicability across diverse systems, underscoring its ability to provide accurate and reduced architectural representations. Our approach allows for a more in-depth understanding of the system's structure and behavior while managing the intricacies of the source material. This capability is of particular value in large systems with large code bases, where the ability to provide clarity and simplification is important for management and the evolution of the software architecture.

7.2.2. Experiment Results

This section analyzes the experimental results of our view-based approach for automatically reverse engineering software architecture models from technology-induced views. In this step, only the technology-specific extraction rules are applied to the case study to extract the views. This evaluation encompasses three principal domains: syntactic and semantic correctness, scalability, and abstraction capability. Our approach has been integrated into a CI/CD pipeline, facilitating the automated generation and validation of architectural models. The syntactic correctness was initially confirmed by PlantUML and subsequently by the EMF validation framework and the Eclipse environment, thereby ensuring structural and semantic correctness. The scalability of the approach was evaluated by comparing the execution times across the case studies. This demonstrated that the approach exhibited predictable performance as input complexity increased. The capacity of our approach to abstraction was evaluated by examining the process of reducing complex software systems

into concise architectural models. The analysis yielded substantial evidence supporting the approach's efficacy on a large code base.

7.2.2.1. Syntactic and Semantic Validation

In this experimental evaluation step, our approach was integrated into a GitHub action within a continuous integration and delivery pipeline, as previously described. This integration enabled the automated generation and subsequent validation of architectural models, which are now subjected to syntactic and semantic correctness checks.

The textual notation of the UML diagrams was handled by PlantUML, which successfully generated the corresponding diagrams for each case study without any error messages. This outcome indicates that the syntactic correctness of the UML architecture models was successful. The capacity of PlantUML to accurately interpret and visualize the textual descriptions of the UML diagrams confirms the correctness of the model syntax. Each diagram generated was found to conform to the expected PlantUML default, demonstrating that the automated process maintained the correctness of the syntactic structure.

Given the complexity and size of the PCM models, they were validated using the EMF Validation Framework. The EMF Validation Framework offers a powerful process for ensuring the structural correctness of models by checking them against predefined validation rules. To illustrate, one of the constraints says that a provided role must be bound, while another says that every user action except start and stop must have a predecessor and successor. These OCL constraints were applied, intending to further verify the syntactic correctness of the models. These constraints permit the detailed specification of rules that model elements must obey, thereby enabling validation beyond basic structural checks.

The PCM instances were subjected to model validation within the Eclipse environment. This validation process included an examination of potential errors and warnings that could indicate syntactic or semantic issues within the models. The models were successfully validated within the Eclipse environment, with no errors or warnings reported. This suggests that the PCM architecture models adhere to the syntactic rules and meet the semantic criteria necessary for accurate and meaningful representations of the system architectures.

The successful validation of both UML and PCM models illustrates our approach's efficacy in generating syntactically and semantically valid architecture models. The integration with the CI/CD pipeline and the experimental validation steps ensure that the models remain compatible and error-free, thus facilitating their use in further architectural analysis and decision-making processes. Utilizing tools such as PlantUML and the EMF Validation Framework with OCL constraints, provides a mechanism for ensuring model correctness, thereby enhancing the reliability of the reverse-engineered architecture models.

The validation process demonstrated the efficacy of our approach in dealing with large models. The capacity to automatically generate and validate PCM models, which are inherently more detailed than UML diagrams, exemplifies our approach's scalability and

practical applicability. By ensuring that these models are syntactically and semantically valid, we provide a foundation for their use in various analytical and predictive tasks, such as performance prediction and security analysis. The experimental evaluation of our approach has confirmed the syntactic and semantic validity of the reverse-engineered architecture models. Integrating our approach into a CI/CD pipeline with validation using PlantUML, the EMF validation framework, and OCL constraints ensures the reliability of the generated models. These findings evaluate that our approach is practical for maintaining the correctness of architectural models.

7.2.2.2. Scalability Validation

For each end-to-end case study, the GitHub action was executed ten times independently with identical inputs, and the execution times were recorded for each run. The differences in execution times were minimal, with a maximum deviation of one second. To validate the reliability of our findings, we conducted a comparative analysis between the execution times of the GitHub action and those of our local test runs. The scalability of the GitHub action execution times was comparable to the local execution times, indicating the stability of the automated process.

Execution Time	Technologies	Artifacts	LoC	Components
`acmeair/acmeair`				
(4.736 ± 0.058) s	6	106	7426	19
`anilallewar/microservices-basics-spring-boot`				
(3.381 ± 0.038) s	7	107	3859	10
`apssouza22/java-microservice`				
(4.328 ± 0.084) s	5	159	5596	38
`callistaenterprise/blog-microservices`				
(3.512 ± 0.043) s	5	77	2865	14
`cloudscale-project/cloudstore`				
(4.218 ± 0.067) s	4	113	9777	35
`corona-warn-app`				
(16.783 ± 0.254) s	8	1516	380 470	117
`corona-warn-app/cwa-dcc-server`				
(3.811 ± 0.074) s	7	77	4723	20
`corona-warn-app/cwa-log-upload`				
(3.594 ± 0.062) s	6	59	3466	15
`corona-warn-app/cwa-ppa-server`				
(6.164 ± 0.099) s	6	391	21 241	49
`corona-warn-app/cwa-server`				
(9.116 ± 0.200) s	6	856	342 858	10

Execution Time	Technologies	Artifacts	LoC	Components
`corona-warn-app/cwa-testresult-server`				
(3.412 ± 0.051) s	7	48	2889	5
`corona-warn-app/cwa-verification-portal`				
(3.307 ± 0.058) s	6	35	1878	8
`corona-warn-app/cwa-verification-server`				
(3.875 ± 0.044) s	7	85	5121	12
`descartesresearch/teastore`				
(7.100 ± 0.165) s	9	295	21 724	67
`descartesresearch/teastore` (project-specific)				
(10.738 ± 0.356) s	9	295	21 724	53
`ewolff/microservice`				
(3.662 ± 0.044) s	4	70	2366	18
`ewolff/microservice` (project-specific)				
(5.999 ± 0.113) s	4	70	2366	20
`fernandoabcampos/spring-netflix-oss-microservices`				
(3.307 ± 0.030) s	4	55	1551	10
`fudanselab/train-ticket`				
(38.205 ± 1.704) s	9	2006	288 034	204
`georgwittberger/apache-spring-boot-microservice-example`				
(3.114 ± 0.044) s	4	22	491	9
`jferrater/tap-and-eat-microservices`				
(3.277 ± 0.046) s	5	66	1457	8
`kbastani/spring-cloud-event-sourcing-example`				
(4.386 ± 0.050) s	7	476	82 201	46
`kit-recipe-app/recipebackendnew`				
(3.944 ± 0.047) s	5	68	2321	22
`kit-sdq/esda`				
(3.200 ± 0.040) s	5	37	1323	2
`koushikkothagal/spring-boot-microservices-workshop`				
(3.036 ± 0.033) s	3	27	947	9
`mdeket/spring-cloud-movie-recommendation`				
(3.444 ± 0.055) s	5	52	1886	14
`meet-eat/meet-eat-server`				
(4.900 ± 0.109) s	2	86	6211	44

Execution Time	Technologies	Artifacts	LoC	Components
`openmrs/openmrs-core`				
(25.786 ± 0.292) s	9	2266	344 254	51
`openmrs/openmrs-module-webservices.rest`				
(9.025 ± 0.245) s	6	747	58 911	52
`piomin/sample-spring-oauth2-microservices`				
(2.969 ± 0.034) s	3	19	470	4
`rohitghatol/spring-boot-microservices`				
(3.228 ± 0.031) s	6	67	2716	10
`shabbirdwd53/springboot-microservice`				
(3.347 ± 0.033) s	3	37	914	11
`spring-petclinic/spring-petclinic-microservices`				
(3.562 ± 0.079) s	8	123	4320	13
`sqshq/piggymetrics`				
(4.123 ± 0.034) s	5	148	6286	29
`webgoat/webgoat`				
(16.170 ± 0.286) s	8	484	64 391	259
`yidongnan/spring-cloud-netflix-example`				
(3.055 ± 0.062) s	4	41	1046	9

Table 7.2.: The table includes all case studies and their respective scalability metrics. It includes the mean execution time with standard deviation, the number of different technologies, the total number of artifacts and the cumulative number of lines of code across all artifacts in the study, and the number of components in our final model.

Figure 7.2 compares the average execution time for each case study with the input data size, defined as the sum of the LoC for all relevant artifacts analyzed. This graph offers insight into the relationship between the complexity and size of the code base and the time required for automated reverse engineering. Figure 7.3 compares the average execution time for each case study to the number of relevant artifacts analyzed. This metric offers a distinct view of input size by focusing on the number of individual files rather than their cumulative size in terms of LoC. Figure 7.4 compares the average execution time for each case study to the number of elements in the final architecture model. This reflects the output complexity of the reverse engineering process.

In Figures 7.2, 7.3 and 7.4, the execution time is plotted on the logarithmic x-axis, while the input or output size metric is plotted on the logarithmic y-axis. This configuration provides a transparent depiction of the correlation between execution time and input or output size. These graphs illustrate the size of the technology stack for each case study, thereby providing context for the observed execution times. The precise numerical values of these metrics are provided in Table 7.2, offering a basis for the graphical representations.

As anticipated, the average execution time increases with the input data size for both metrics: LoC and the number of artifacts. This trend indicates a direct correlation between the complexity of the code base and the time required for analysis. Consequently, the case study with the longest execution time is the one that requires the most LoC to analyze. This finding serves to illustrate the impact of code volume on processing time. Conversely, the case studies with the shortest execution times have the fewest artifacts to analyze. This demonstrates the potential for efficiency gains in scenarios with smaller input sizes.

Figure 7.4 provides further insight into the relationship between execution time and the complexity of the final architectural model. The execution times of case studies with larger components in their final models tend to be longer, reflecting the additional processing required to represent architectural details accurately. This observation agrees with the assumption that the initial complexity, quantified by the number of components, also affects the execution time.

The analysis demonstrated that case studies with comparable input or output sizes exhibited comparable execution times. This indicates that the automated approach scales predictably with the input data size and the resulting models' complexity. In particular, case studies with the same number of components in their final architecture models exhibited comparable execution times, reinforcing that output complexity is a determinant of processing time.

The `fudanselab/train-ticket` case study, comprising 204 components, 2006 source files, and a total of 288 034 LoC, was extracted in (38.205 ± 1.704) s, exhibiting the longest execution time among the case studies. This case study employed 9 technologies and showed the greatest variability in execution time, with a standard deviation of more than one second. All other standard deviations were less than one second. The `piomin/sample-spring-oauth2-microservices` case study comprising 4 components, 19 source files with a total of 470 LoC, and three technologies, exhibited the shortest execution time. The extraction was completed in a mere (2.969 ± 0.034) s. The remaining numbers for the case studies are presented in Table 7.2.

The comparative analysis between the GitHub action and local test runs, coupled with the detailed experimental evaluation of execution times across different input sizes and output complexities, illustrates the scalability of our approach. An evaluation of execution times, correlated with input and output metrics, has been conducted to provide a detailed experimental evaluation of the scalability of our automated reverse engineering approach. These results offer insights into the practical implications of automated software architecture modeling tools, particularly regarding scalability and processing efficiency across large code bases.

7.2.2.3. Abstraction Capability Validation

To evaluate the abstraction capability of our approach for larger systems, we compared the number of source files and total LoC to the number of components in the final architectural model. This ratio serves as a indicator of the applicability of our reverse engineering

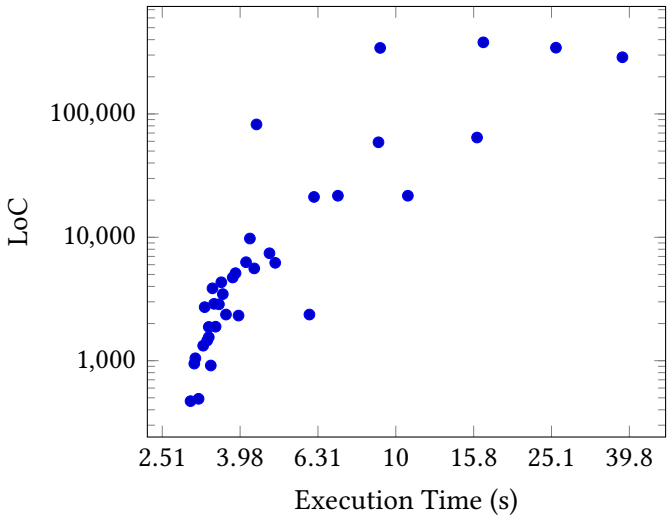


Figure 7.2.: The graph depicts the ratio of the average execution time in seconds to the sum of the LoC for all artifacts analyzed in the case study. The precise numerical values can be found in Table 7.2.

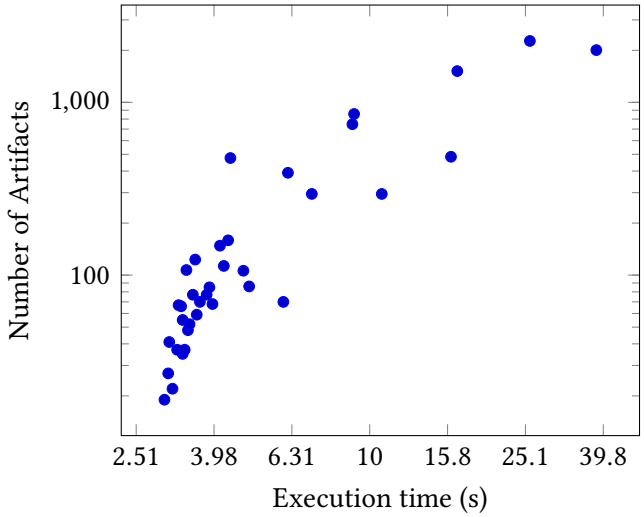


Figure 7.3.: The graph depicts the ratio of the average execution time in seconds to the total number of artifacts analyzed in the case study. The precise numerical values can be found in Table 7.2.

approach to reducing the complexity of the original software systems. By maintaining a lower ratio of components to input size, we demonstrate that our approach can accommodate large systems without overburdening the resulting architectural models with excessive detail [Lan19]. This abstraction capability is important in practical applications, where software systems often comprise thousands of files and millions of LoC.

The total number of source files was calculated as the sum of the values presented in Table 7.1, which detailed these metrics for each system under study. Figure 7.5 depicts the ratio of the number of source files to the number of components in the final model. Figure 7.6 depicts the ratio of total LoC to the number of components. In both graphs, the input size, as measured by the number of files or LoC, is plotted on the x-axis, while the

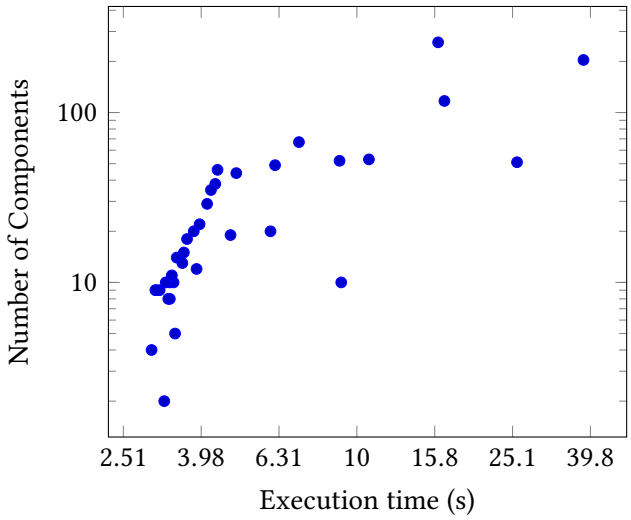


Figure 7.4.: The graph depicts the ratio of the average execution time in seconds to the total number of components in the final architectural model. The precise numerical values can be found in Table 7.2.

number of components is plotted on the logarithmic y-axis. This visual representation provides a clear and intuitive understanding of the relationship between the size of the input and the complexity of the final architectural model. These results are precious in understanding our approach's abstraction capability.

The results of this analysis indicate a positive correlation between the number of source files and the number of components. This trend is observed across all case studies, suggesting that larger code bases naturally lead to more complex architectural models. The ratio of LoC to the number of components provides further insight into the abstraction capability of our approach. As the number of LoC increases, the number of components increases, albeit much slower. This evidence indicates that while the initial code base may be voluminous, our approach abstracts this information into smaller architectural components. The logarithmic scale employed in Figures 7.5 and 7.6 illustrate our approach's efficacy. This is evidenced by the observation that the number of components is much less steep than the growth in the input size.

The `corona-warn-app/cwa-server` case study, consisting of ten components, 856 source files, and a total of 342 858 LoC, has the largest abstraction ratio of 85.60 from our case studies. In contrast, the `webgoat/webgoat` case study, consisting of 259 components, 482 source files with a total of 64 391 LoC, and eight technologies, has the smallest abstraction ratio of 1.86 from our case studies. This substantial reduction demonstrates our approach's efficacy in simplifying the structures of complicated software systems into understandable architectural models.

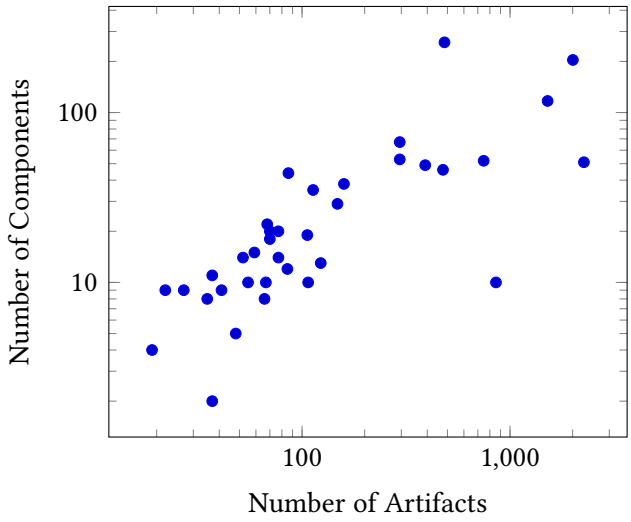


Figure 7.5.: The ratio of source files to the number of components in the architectural model indicates the level of abstraction. The precise numerical values can be found in Table 7.2.

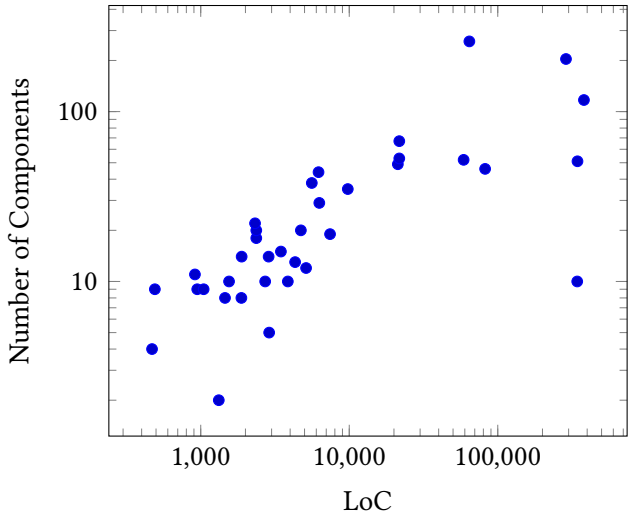


Figure 7.6.: The ratio of the LoC to the number of components in the architectural model indicates the level of abstraction. The precise numerical values can be found in Table 7.2.

7.3. Accuracy Validation

Ensuring the accuracy of our Retriever approach is essential to facilitating its implementation in professional software analysis. This section outlines our steps to demonstrate the accuracy of our software architecture reverse engineering and vulnerability annotation. A curated set of gold-standard case studies, developed through manual analysis of detailed architectural documentation, was used as a benchmark for evaluation. The validation process includes technology-specific and project-specific validations utilizing quantitative metrics such as precision, recall, and F_1 score to measure the agreement between automated and manual models. The vulnerability annotations were evaluated within a CI/CD pipeline,

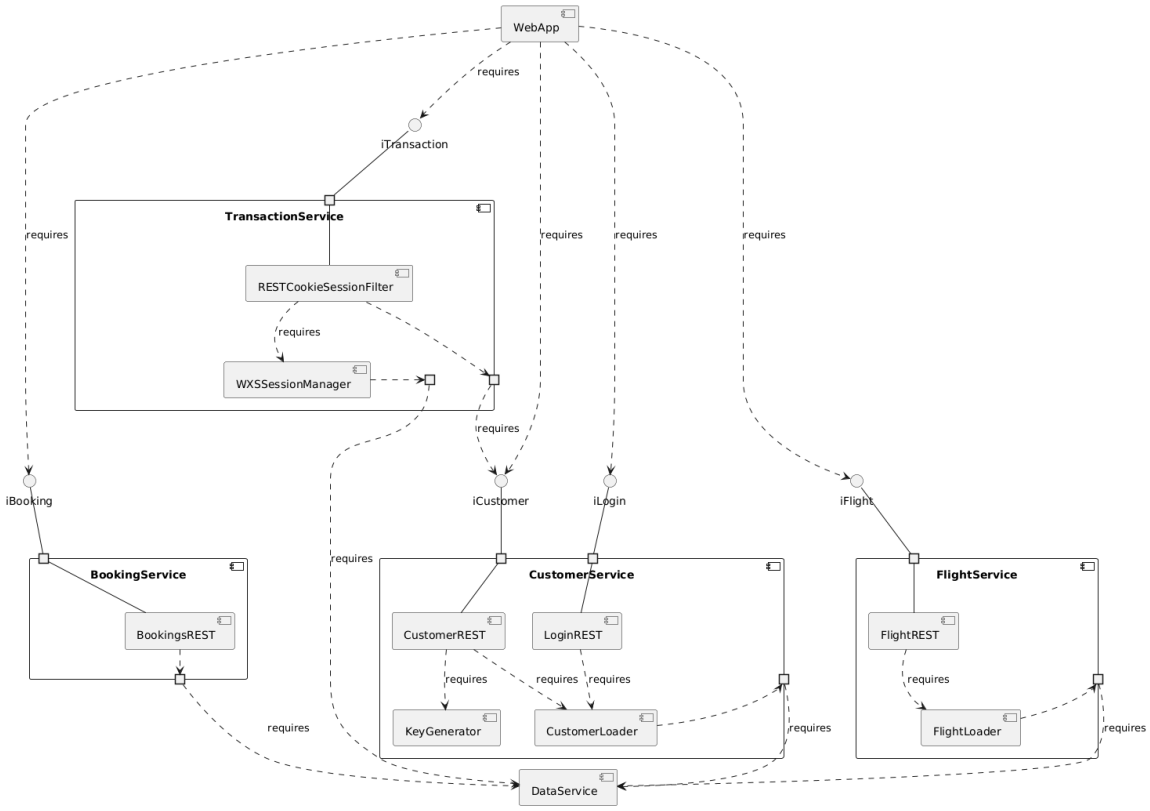


Figure 7.7.: The component diagram of the Acme Air system. The architecture comprises several components: the common module, loader, service module, web application, and driver.

reflecting the characteristics of real-world development environments. This multifaceted approach, further refined based on expert feedback, ensures that our approach accurately captures architectural components and reliably identifies vulnerabilities. The resulting analysis demonstrates our approach's accuracy and applicability to complex software systems.

7.3.1. Gold Standard Case Studies

To validate the accuracy of our approach, we performed experimental validation on a subset of the previously used case studies. The rationale is that a gold standard needs to be established for these case studies, against which our results can be evaluated. A gold standard was established for the following seven case studies, which describe the software architecture of the case studies. In addition, the vulnerability views for four of the case studies were validated. Two project-specific rules were created and validated for two case studies, and the results were compared to those obtained using technology-specific rules. The following seven case studies were selected for accuracy validation:

Acme Air System [TS24] is a component-based software system developed as a sample application to demonstrate the structure and scalability of a fictitious airline system.

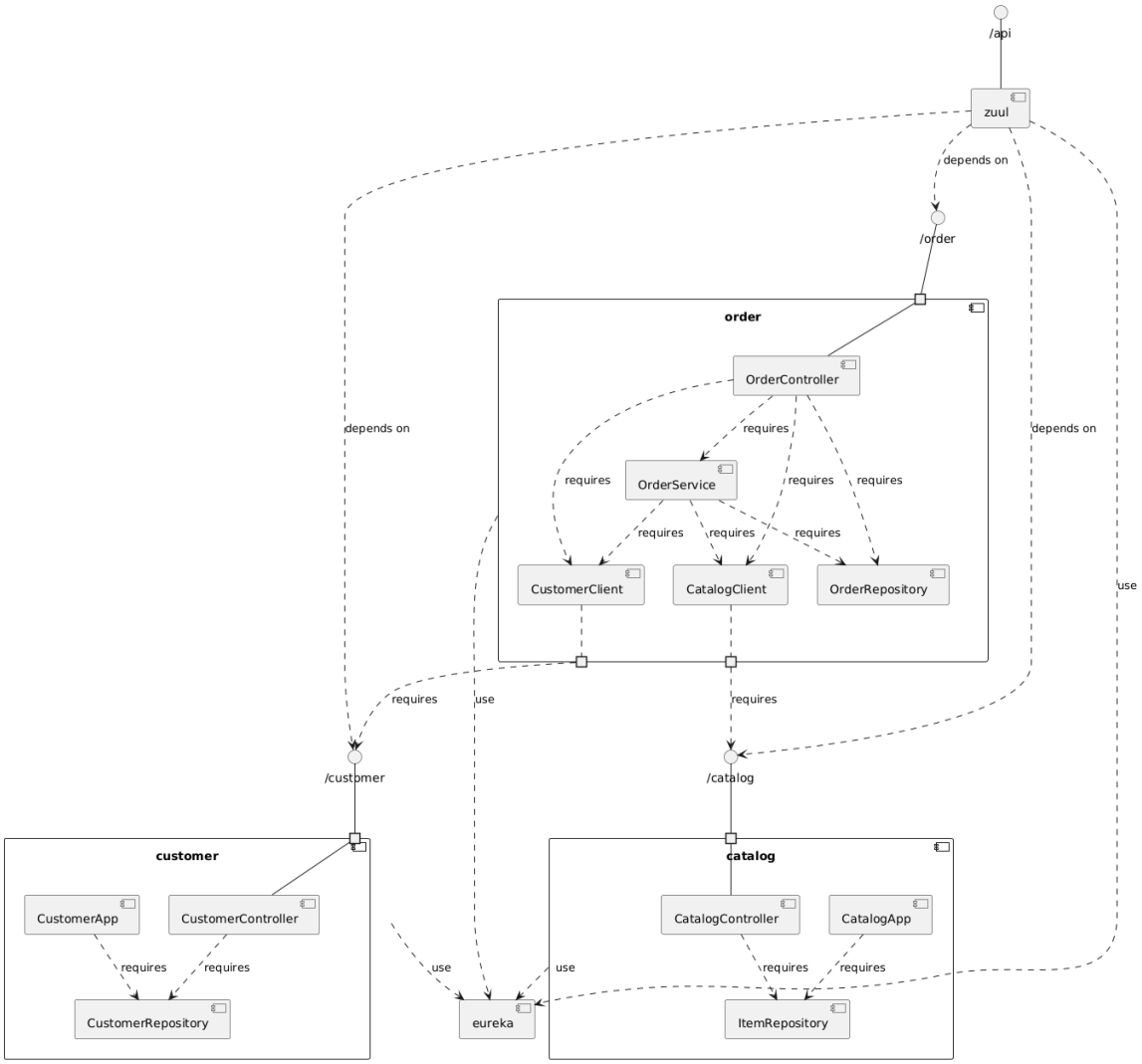


Figure 7.8.: The component diagram of the Microservice Sample system. The architecture consists of multiple microservices, each encapsulating its functionality and data.

It is designed to handle a high volume of web API calls and supports scalability for billions of daily transactions. Its architecture is structured to be deployed on public cloud infrastructures and can support multiple user interaction channels, emphasizing mobile and web-based interfaces. Figure 7.7 illustrates the component diagram of the system, showing the components and their interactions. The architecture consists of several components. The common module contains Java entities that act as shared resources across different application parts, ensuring consistency and reuse of data structures. The loader is responsible for loading data into the system's data store, facilitating initial setup and data population for testing purposes. The service module defines the data service interface, providing a standardized way for different components to interact with the data layer. There are two specific implementations of this data service layer. The web application serves as the user interface, providing a web application with accompanying Java REST services to

enable end-user interaction. Finally, the driver component contains workload driver scripts and supporting resources necessary for benchmarking and performance testing. The structure demonstrates a scalable approach to component-based software architecture in a cloud-deployed environment.

The Spring Boot Microservices Template [All24] is an example of a microservices-based software architecture implemented using the Spring Boot and Spring Cloud frameworks. It is structured around loosely coupled, independently deployable components, each performing a specific task. The Config Server component acts as a centralized configuration manager, enabling the management of configuration properties for other services in the system. The Service Registry facilitates service discovery, allowing microservices to discover each other dynamically. The system includes dedicated microservices for handling domain functionalities such as user, task, and comment, each representing a separate bounded context within the architecture. These services communicate via REST APIs and are designed to scale independently. The Auth Server component implements authentication and authorization, ensuring secure interactions between the microservices. The API Gateway is a single entry point, routing external requests to the appropriate microservices and handling issues such as load balancing, authentication, and rate limiting. The web portal component provides a user interface layer that interacts with the backend microservices through the gateway. A tracing server is integrated to provide distributed tracing, enabling performance monitoring and analysis across the various microservices. This architecture exemplifies best practices for building scalable and maintainable microservice systems.

The Tea Store System [Kis+24; Eis+20; Kis+18] is a microservices-based reference application designed to demonstrate the principles of component-based software architecture in the context of an online store. It is structured around multiple independently deployable services that communicate via REST APIs. The central component, the web interface, serves as the user interface, handling HTTP requests and providing functionality such as product browsing, shopping cart management, and order processing. This component relies on backend services to retrieve and display relevant data. The Auth Service handles authentication and authorization, providing secure access control by managing user sessions and restricting specific actions to authenticated users. The Persistence Service is the data storage backbone, managing persistent data across entities such as products, users, orders, and shopping carts. It ensures data correctness and consistency throughout the system. The Image Service is dedicated to storing and serving product images, facilitating the visual representation of products in the user interface. The Recommender Service optimizes the user experience by analyzing behavioral data and purchase history to generate personalized product recommendations. Finally, the load balancer component distributes incoming requests across multiple instances of services to optimize resource utilization, improve scalability, and ensure high availability. This component-based architecture exemplifies modular design, allowing for independent development, deployment, and scaling of individual services within the Tea Store system.

The Microservice Sample [Wol24] is a component-based software system that exemplifies the microservices architectural style, demonstrating a design where individual services function as independent components, each addressing a specific business capability. Figure 7.8 illustrates the component diagram of the system, showing the components and their interactions. The components consist of multiple microservices, each encapsulating its functionality and data. The Order Service manages orders and provides order creation, update, and retrieval capabilities. It interacts with other services, such as customer and product services, to perform its operations, demonstrating service collaboration within the system. Customer Service manages customer-related data and operates independently, adhering to data encapsulation and autonomy principles. The product service maintains product data, including inventory management and product details. In addition, the Catalog Service aggregates product information and integrates with the Product Service to present product listings to external clients. These services communicate via REST APIs, ensuring interoperability while maintaining loose coupling. Supporting components include an API gateway that serves as a single entry point for client requests and routes them to the appropriate services and a discovery server that facilitates service registration and discovery. Each microservice maintains its own database, reflecting decentralized data management and supporting the microservice architecture's principle of independent scalability and deployment.

The Apache Web Server and Spring Boot Microservice Example [Wit24] is a component-based software system that demonstrates the integration of multiple microservices to compose cohesive web pages. The architecture consists of three independent microservices developed using Spring Boot, each responsible for different functionality. These microservices interact through an Apache web server that acts as a reverse proxy, handling request routing and response integration. The content service provides the Hypertext Markup Language (HTML) structure, while the product service provides product-related data in HTML and JSON formats. The cart service manages the shopping cart functionality and provides a JavaScript library for client-side behavior. The system uses both server-side and client-side composition techniques. Server-side composition ensures integration by assembling HTML content via server-side includes, while client-side composition supplements the user experience using JavaScript. This approach allows the system to seamlessly adapt to content needs while maintaining the appearance of a unified application. This architecture combines multiple web services into a single, integrated commerce application and highlights the balance between modularity, performance, and maintainability.

The Movie Recommendation System [Dek24] is an open-source, microservices-based software system developed using Spring Cloud technologies. It consists of six microservices, each with a specific responsibility. The Discovery Service acts as a service registry, enabling the discovery of services within the architecture. The Config Service provides centralized configuration management and must be the first service to start, ensuring consistent configuration across all components. The User Service manages user-related data using a MySQL database for persistence, where user details are stored and retrieved. The Movie Service handles movie data and interactions,

using a database as a data store to manage movie information. The Recommendation Service implements recommendation logic using a Neo4j graph database that stores relationships between users, movies, and their interactions. This service generates recommendations based on user preferences and behaviors. Finally, the recommendation client is an API gateway that interfaces with all the other services. It uses a load balancer for client-side load balancing, distributing requests across multiple service instances, and using a circuit breaker service to maintain resiliency in the face of service failures. This component-based architecture demonstrates the use of various technologies to provide a scalable and resilient recommendation system.

The Spring Pet Clinic System [RSD+24] is an example of a microservices-based, component-based software architecture typical of modern software development. It is organized into microservices, each encapsulating a specific business capability related to veterinary clinic operations. Customer Service manages pet owner information and provides an API for interaction with other services. The Veterinarian Service is responsible for managing veterinarian information, maintaining records, and enabling the retrieval and management of veterinarian-related data. Similarly, the Visits service is responsible for scheduling and tracking pet visits, storing visit details, and maintaining pet and veterinarian associations. The Pets service handles pet-related data, keeping information about each pet, such as type, name, date of birth, and associated owner, and interacting with customer service. The API Gateway acts as a central access point, managing request routing, load balancing, and system security, while the Discovery Server facilitates service registration and discovery, supporting scalability. The Configuration Server provides consistent and centralized configuration settings management across all microservices. Each service communicates via REST APIs, maintaining loose coupling and data independence, as each typically uses its own database. This architectural design ensures scalability, fault tolerance, and maintainability and demonstrates a practical application of component-based software architecture principles in a microservice environment.

The Piggy Metrics System [Luk24] is a financial tracking software system that uses a microservices-based architectural design and component-based approach, which aligns with modern software design principles. The system consists of three microservices, each operating as an autonomous component performing a discrete function. The Account Service manages user account information, including financial goals, savings, and spending preferences. It provides the basic operations for creating, updating, and retrieving user-specific data and acts as a central repository that interacts with other services to maintain data consistency. The statistics service aggregates and processes financial data relating to individual users. The system can calculate metrics, including total savings, spending, and overall financial status, by retrieving information from the Account Service. This service gives users insight into their financial behavior and stores the processed data for subsequent reporting. The Notification Service manages user notifications and alerts, using the Account and Statistics Services data to generate reminders and messages related to savings goals and spending limits. Additional infrastructure services are also provided to support these components. These include a Spring Cloud Config server for centralized configuration management, an API

gateway for request routing and load balancing, and a service discovery server. The modular, loosely coupled architecture allows services to be developed, deployed, and scaled independently.

7.3.2. Experiment Design

To accurately assess the precision of our approach, we first construct a reference model of the software system under consideration. This model serves as a basis for comparing the results of our automated extraction process with those obtained by manual analysis. This evaluation entails studying the source code and configuration files to identify individual components and delineate their respective functionalities within the system. This is followed by a direct comparison between the model generated by our approach and any existing documentation or architectural diagrams available in the software repository or the associated documentation. Such a comparative analysis is of great importance for the refinement of the accuracy of our model and the assurance of its alignment with the understanding of the software's architecture.

Understand the System The initial phase of the investigation examined the system's objectives, operational capabilities, user interactions, and tasks. This was accomplished by analyzing the available documentation, source code, design documents, and other publications. The objective was to understand the system and identify components by applying different analytical techniques and resources.

Architectural Style Selection Once the system's operational requirements were identified, an architectural style was selected to meet those requirements best. This entailed delineating specific rules and conventions about the components' advancement, distribution, and operation. In addition, the syntax and semantics of interfaces were defined, along with specifications for lifecycle management, deployment procedures, and runtime environments.

Component Diagram Construction Subsequently, component diagrams were employed to visually represent the identified components, including their interactions, interfaces, dependencies, and connections. Standard notations, such as UML or occasionally custom schemas, ensured clarity, consistency, and ease of understanding. Tools such as PlantUML were employed to visualize the system architecture.

Architectural Pattern Analysis This phase entailed an assessment of prevalent software architecture patterns, including n-tier, client-server, and microservices. This research determined how these patterns could improve system characteristics and stability. Subsequently, the selected patterns were integrated into the defined component model and architectural style to enhance system functionality.

Verification and Refinement The final stage of the process involved collaboration with relevant stakeholders to validate the architectural model. Feedback was sought to ensure that the model remained aligned with the evolving requirements and constraints of the system and to mitigate potential biases and inaccuracies.

Stakeholder Engagement A collaborative effort was undertaken with software developers from other case studies to review and discuss the architectural model. This collaboration was important for refining the architecture, ensuring alignment with industry practices, and reducing bias by incorporating feedback.

Documentation Engagement A review of existing architectural documentation and diagrams was conducted to identify and resolve inconsistencies. This comparison ensured that our architectural model was closely aligned with the documented intent and original design principles, thereby improving the model's accuracy and reducing inherent bias.

Creating a manual software architecture model is a gold standard for validating our automated reverse engineering approach. Manual development is essential for establishing a representation of component-based software architectures. This process is a benchmark for assessing the accuracy of automatically generated models. The process employs an iterative approach designed to encapsulate the architecture of the software systems under review. This process is applied to a selected subset of case studies. The selection criteria for these case studies hinge on the availability of architectural documentation. Such documentation is indispensable for manually constructing architectural models, as it provides insights into the system's structure, including identifying components, their interactions, interfaces, dependencies, and connections. This documentation serves as a foundation for ensuring that the manual models accurately reflect the actual architectural structure of the software systems.

Our approach entails a comparison of the manually created UML models to the existing architectural diagrams provided within the case studies. This comparison is essential to mitigate biases and align the manual models closely with the documented architectural intents. To improve the accuracy of these models, discussions are held with software developers who are part of other case studies. These discussions are conducted to refine the gold standard model to ensure that it reflects theoretical accuracy and aligns with real-world software development practices. Through these iterative steps and collaborative refinement processes, the manual architecture model evolves into a standard that guides and validates the outcomes of our approach. This iterative validation and refinement ensures that our approach remains aligned with scientific rigor and industry relevance, providing a solid foundation for further research and development in reverse engineering software architecture.

The development of gold standards, independent of our Retriever approach, is crucial for validating the accuracy of technology-specific extraction rules. First, these rules are implemented and applied to generate software architecture models in various case studies. Next, gold standards are manually created through detailed analysis to serve as our benchmarks. In the next phase, project-specific extraction rules are formulated based on the insights from comparing the reverse-engineered technology-specific models with the manually developed gold standards. This process refines the project-specific rules to improve our automated model extraction, ensuring an accurate representation of the underlying architecture. For the second evaluation goal, we focus on three steps: validating the accuracy

of the technology-specific extraction rules in capturing the architecture (Section 7.3.2.1), integrating new project-specific rules to improve accuracy (Section 7.3.2.2), and evaluating the effectiveness of our automated reverse engineering in identifying vulnerabilities by comparing our annotations with those of a security tool (Section 7.3.2.3).

7.3.2.1. Technology-Specific Validation

The second goal of our evaluation plan is to create a manual software architecture model that can be used as a benchmark for evaluating the accuracy of the architecture models generated by our approach. This crafted reference model encompasses all anticipated components, delineating their interactions, interfaces, dependencies, and connections essential to the studies under consideration. This step commences with directly comparing the manual model with the results of our automated reverse engineering approach. Such a comparison is essential for answering specific evaluation questions related to the accuracy of our automated processes and is part of our overall evaluation plan. By performing these comparisons, we can assess how our automated models match the manually generated gold standards and identify areas where the automated approach may be deficient.

An analysis of the structural and behavioral properties of the manually identified component architectures and those inferred by the automation is conducted to perform this comparison. The criteria for comparison include:

Component Boundaries and Groupings The delineation and grouping of components within the system are evaluated by both approaches.

Component Interfaces The interfaces provided and required by components, including their specified URIs, are compared to ensure they match both models.

Component Calls and Relationships The relationships and external calls between components are evaluated per the SEFF.

Metrics such as precision, recall, and F_1 score facilitate quantitative analysis. Precision is defined as the degree of accuracy with which the elements identified by our approach correspond to those in the manual model. In contrast, recall is the completeness of our approach, indicating the proportion of elements identified manually that were also captured by our approach. Finally, the F_1 score provides a harmonic mean of precision and recall, balancing both measures. In an ideal scenario where recall and precision reach 1.00, the F_1 score would be 1.00, indicating a perfectly accurate architecture model.

In the context of our evaluation, true positives represent model elements that have been identified in both the manual and automated models. This indicates that the automated process has successfully captured these elements. False positives occur when elements are determined solely by the automated process. In contrast, false negatives represent instances where the automated process fails to identify elements noted in the manual analysis. Each case study is subjected to a comparison process. This provides an evaluation of the accuracy of our reverse engineering approach. The results of each comparison are categorized and analyzed to calculate the metrics mentioned above, thereby providing a framework for

evaluating the accuracy of our reverse engineering approach across multiple case studies. This comparison illuminates our automated approach's strengths and identifies potential shortcomings that can inform future improvements and refinements.

7.3.2.2. Project-Specific Validation

In the subsequent evaluation phase, we will improve our reverse engineering process by optimizing and integrating new project-specific rules. This second step is essential in determining how these custom rules enhance the accuracy of our automated model reconstruction efforts. This evaluation step aims to assess the accuracy of the architecture models after incorporating the new, specific rules and determine the effort required to formulate and apply these rules accurately to capture recurring project-specific patterns.

A reference model for the software system under study was first constructed to facilitate this evaluation. This model serves as a reference point against which the results of our automated reverse engineering process are compared. This comparison aims to identify inconsistencies and areas where the model reconstruction could be improved by employing additional guidelines. The evaluation process entails applying a predefined set of technology-specific rules to the system under study. The resulting architectural model is then compared to the expected model to assess its accuracy.

The efficacy of the newly formulated project-specific rules is determined by employing the harmonic mean of the precision, recall, and F_1 scores. Precision measures the accuracy with which items are correctly identified as relevant. In contrast, the recall metric measures the model by identifying any omissions. True positives are items that are correctly identified as relevant. False positives are items that are incorrectly included in the generated model. Relevant items that are not included in the generated model are referred to as false negatives.

It is important to note that a high recall does not necessarily indicate a high degree of accuracy, as it may include misidentified items. Similarly, precision is of great importance in that it avoids misidentifications. The extracted software architecture is analyzed after the comparison to identify potential improvement areas. This may necessitate integrating novel rules into the existing rule set to address specific complexities identified during the evaluation process.

The quantity of newly formulated project-specific rules and the total LoC involved in their definitions quantify the model's accuracy and the effort required to define these new rules. In the ideal scenario, if the existing rules are perfect, both metrics approach zero, indicating that no additional coding or rule definition is required for accuracy model recovery. This phase tests the rule set's accuracy and identifies areas for future improvements that will further refine the reverse engineering process.

7.3.2.3. Vulnerability Validation

Our second evaluation question for the second goal is to assess the accuracy of our automated reverse engineering for vulnerabilities with our approach. This evaluation was conducted within the operational framework of a CI/CD pipeline hosted on GitHub. This setting reflects real-world software development practices and improves the relevance of the results. In this context, the concurrent development of architecture and vulnerability models improved the multidimensional analytical power of our approach. Specifically, the output of Snyk, a security tool that annotates each project with potential vulnerabilities, was employed to compare the annotations to the vulnerability models generated. For this analysis, only vulnerabilities assigned a CVE identifier were considered. This approach ensured the evaluation focused on known and security threats.

The comparison process yielded several findings, which are presented below. The identification of true positives was achieved when both the PCM vulnerability annotation and the corresponding textual description from the Snyk CLI reported the same CVE, indicating accurate vulnerability detection. False positives were identified when the model predicted a vulnerability not confirmed by the textual output from the Snyk CLI. This suggests that the prevalence of security threats may have been overestimated. False negatives were identified when the textual output from the Snyk CLI listed a vulnerability not present in the corresponding PCM vulnerability model. This indicates that potential security issues may have been underestimated.

We compiled the results to calculate the precision, recall, and the F_1 score. Each of these metrics provides valuable insight into the accuracy of our approach. Precision offers a measure of the accuracy of identified vulnerabilities, which is essential to avoid the generation of unnecessary alerts that could result in the waste of resources. Recall assesses the system's ability to identify all relevant vulnerabilities critical to ensuring security coverage. The F_1 score is a statistical tool that balances precision and recall, providing a single metric to evaluate the trade-offs between identifying as many vulnerabilities as possible and minimizing false positives.

This validation aims to substantiate the accuracy of the automated reverse engineering approach in identifying vulnerabilities within software architectures through a detailed evaluation. This will provide valuable insights into the approach's accuracy in security contexts.

7.3.3. Experiment Results

This subsection presents the experimental results of our approach to reverse engineering software architecture models and vulnerability annotation. The evaluation includes technology-specific validation, project-specific validation, and vulnerability validation across multiple case studies. To assess the performance of our approach, we used precision, recall, and F_1 score metrics and compared the generated models to gold standard architectures. The analysis reveals several categories of errors that provide insights into

the strengths and limitations of our approach. The experimental results highlight that while our approach demonstrates high precision and recall in identifying components and their interactions, it is susceptible to specific errors that affect the overall accuracy of the generated models. Addressing these error categories is essential for refining the extraction rules and improving their accuracy for software architecture and vulnerability analysis.

The error categories identified in our experiments are false positives and false negatives. False positives occur when the approach erroneously includes irrelevant components in the architectural model. This consists of misidentifying placeholder classes or stubs as components, resulting in an inaccurate representation of the system architecture. Redundant technical specifications and irrelevant component categorizations contribute to these inaccuracies, complicating the model with unnecessary detail.

Conversely, false negatives result from omitting components and their associated relationships. Infrastructure elements such as configuration servers, service registries, and interfaces are sometimes overlooked, resulting in incomplete architectural models that fail to capture the entire software system. These omissions hinder the accurate representation of system dependencies and communication paths, undermining the applicability of reverse-engineered models for architectural analysis.

Another category is interface and relationship mapping errors. Misrepresentation or mislabeling of interfaces leads to incorrect associations between components, compromising the accuracy of the architectural model. Similarly, incomplete or inaccurate relationship mappings between components disrupt the model's structural correctness and obscure the true interactions and dependencies within the system.

Finally, duplication errors, where components or vulnerabilities are annotated redundantly, affect the accuracy of the vulnerability validation process. While such duplications do not affect the overall analysis, they introduce noise that complicates the interpretation of vulnerability data.

7.3.3.1. Technology-Specific Validation

The second evaluation goal is to accurately capture the architecture of each reference system, including its components, roles, and data and control flow between components. The extraction process involves applying a technology-specific set of rules to each system and comparing the resulting architecture with the expected architecture. This analysis compares the gold standard model to the model generated by our automated reverse engineering approach.

The comparison results are then classified to address the questions about the second objective of the evaluation plan. The accuracy of the extracted architectures can be evaluated by categorizing these results. The metrics calculated for each case study include precision, recall, and F_1 score. The detailed comparison results are presented with the calculated metrics for each case study. Table 7.3 illustrates the extent to which our automated approach accurately reflects the reference architecture. A high precision value

Structural			Behavioral		
Precision	Recall	F_1	Precision	Recall	F_1
`acmeair/acmeair`					
0.79	0.73	0.76	0.79	0.55	0.65
`anilallewar/microservices-basics-spring-boot`					
0.90	0.64	0.75	0.88	0.78	0.82
`DescartesResearch/TeaStore`					
0.84	0.82	0.83	0.75	0.30	0.43
`ewolff/microservice`					
0.78	0.82	0.80	1.00	0.82	0.90
`georgwittberger/apache-spring-boot-microservice-example`					
1.00	0.89	0.94	0.63	0.71	0.67
`mdeket/spring-cloud-movie-recommendation`					
0.80	0.73	0.76	0.86	0.63	0.73
`spring-petclinic/spring-petclinic-microservices`					
1.00	0.76	0.87	0.94	0.83	0.88
`sqshq/piggymetrics`					
0.89	0.91	0.90	0.69	0.85	0.76

Table 7.3.: The comparison results and the metrics for the approach's accuracy in generating structural and behavioral properties using only the technology-specific extraction rules are presented.

indicates that most of the components and connections identified by our approach are correct. A high recall value suggests that our approach identifies most components and connections in the gold standard. The F_1 score provides a balanced view of both aspects, thereby ensuring the evaluation of the accuracy of our approach.

The application of these metrics enables the quantification of the accuracy of our view-based approach to capture the architecture of software systems. The detailed results for each case study highlight our approach's strengths and limitations, thereby providing valuable insights for further refinement and improvement. These metrics validate our approach's accuracy and identify areas where the technology-specific rule sets and extraction processes can be optimized for improved accuracy.

Acme Air System A comparison of the gold standard model and the reverse-engineered model for the Acme Air system reveals both similarities and differences in component identification and relationship mapping. Our approach is applicable in certain areas, as evidenced by the consistent identification of several components in both models. In particular, the reverse-engineered model accurately identifies components such as booking services and customer loaders. It also correctly identifies the relationships between these components, including the dependency of customer loaders on customer services and the dependency of REST bookings on the booking service. This agreement indicates that our approach is an accurate means of identifying components and their direct interactions. False positives indicate instances where the reverse-engineered model incorrectly

identifies certain elements as standalone despite their relationship to other components. For example, the Acme Air configuration, loader, and loader REST are misidentified as standalone components. This misidentification results in incorrect relationships, such as the configuration linked to the booking service, that may not be consistent with the intended architectural structure. The false negatives show that many components and their relationships, as depicted in the gold standard model, are missing from the reverse-engineered model. In particular, several components, including the web application, key generator, data service, and transaction service, are missing from the reverse-engineered model. In addition, the associated interfaces and dependencies are also missing. The absence of the web application component and its interactions with interfaces such as booking, customer, flight, login, and transaction suggests that our approach may have limitations in discovering specific high-level components and their service interfaces. The absence of the data service component and its connections to booking REST, customer loader, flight loader, and session manager indicates the system's representation of data handling aspects is incomplete. It is clear that relationships such as booking REST, which requires integrating data services and web applications with different interfaces, are essential from an architectural view. However, these relationships are not represented in the reverse-engineered model. Although the reverse-engineered model can identify specific components and their direct relationships, it also exhibits inaccuracies in component misidentification and the omission of components and interactions. Component misidentification occurs when elements are mistakenly treated as standalone components rather than as parts of multiple components. Such misclassification leads to an inaccurate representation of the system's structural hierarchy, leading to incorrect relationship mappings. The lack of components and relationships in the reverse-engineered model is evidenced by the absence of components and their interactions, especially those related to high-level application logic and data services. This omission results in an incomplete architectural representation that may miss system dependencies and communication paths.

Spring Boot Microservices Template When examining the Spring Boot microservices template, it is clear that the gold standard and reverse-engineered models have similarities and differences in their architectural representations of the software system. The reverse-engineered model accurately identifies the components, including the task web service, comment web service, user web service, and authentication server. The internal components of these services, including the task controller, comment service, comment controller, user controller, and authentication user controller, are correctly mapped. In addition, the internal relationships between these components are accurately represented. For example, the dependency of the task controller on the comment service is evident in both models. However, the reverse-engineered model is deficient compared to the gold standard in several respects. Specifically, the model lacks several components, including the absence of several components. These components include the configuration server, the web service registry, the web portal, the API gateway, and the connection between these entities. The reverse-engineered model is deficient in that it lacks task and user service interfaces. While the user task interface is present, its linkage to the general task web service interface is incorrect, as is the use of the root interface for internal relations.

Such discrepancies can be classified as interface mapping errors. In addition, the reverse-engineered model misrepresents certain internal relationships by misusing interfaces, a classification of relationship inaccuracy. To illustrate, the root interface is present but misused for internal and related relationships, which is inconsistent with its intended use in the gold standard.

Tea Store System A comparison of the gold standard model and the reverse-engineered model for the tea store system reveals both similarities and differences. The gold standard model includes many elements, including the web user interface in the front-end package, the database in the database package, and various services such as the authentication service, registry service, image service, persistence service, and recommendation service in the service package. These components are connected by interfaces, including the web user interface, persistence, shopping cart authentication, and load balancer, with explicit data flow relationships. A comparison of the reverse-engineered model with the gold standard model shows that the various components correspond to those in the latter. The auth cart REST and auth user actions REST components are aligned with the authentication service. The index and associated servlets are analogous to the web user interface. These accurate identifications are the real positives of the analysis. However, there are discrepancies between the two models that can be categorized into different defect classes. These defect classes highlight areas where our approach differs from the gold standard model. The problems can be attributed to inappropriate component categorization, including redundant technical specifications, inaccurate interface definitions, the exclusion of components and interconnections, misrepresenting messaging components, and a lack of data flow mapping. These errors must be addressed to improve the accuracy of the reverse-engineered model and align it with the intended architectural framework of the software system. Component composition errors occur when the reverse-engineered model incorrectly groups components into composite structures that do not reflect the logical organization of the system. This misrepresentation leads to false positives where the grouping is inconsistent with the intended architectural structure. Incorporating technical implementation details, such as those related to the ready REST component, into the reverse-engineered model can produce false positives by introducing unnecessary complexity. Typically associated with the technical implementation, these details should not be exposed at the architectural level. Misrepresentation of interfaces can result from inaccuracies in how they are represented, leading to misinterpretation of the interface. This category of defects contributes to the generation of false positives by misrepresenting the interfaces of components. The absence of several components, including database, image, persistence, recommendation, and registry service, in the reverse-engineered model indicates the absence of components and relationships. The relationships and data flows between these components are missing. These omissions are false negatives, indicating that the approach failed to discover system components. The reverse-engineered model cannot accurately represent the data flow between components, as shown in the gold standard model. For example, the flow from the registry service to other services is missing. This omission contributes to false negatives and indicates a failure to represent the interactions between components accurately.

Microservice Sample A comparison of the gold standard with the reverse-engineered model for the microservice example reveals both similarities and differences. Both models correctly identify the components of the software systems. Components, including the order controller, order service, customer client, catalog client, and order repository, are present in both models. However, the reverse-engineered model uses fully qualified class names that reflect the package structure. The internal relationships between these components are represented in a manner that is consistent across both models. In addition, both models illustrate the representation of dependencies, such as the order service's dependencies on the customer client and catalog client and the associations between controllers and repositories in the customer and catalog components. The models include interfaces such as order, catalog, and customer and their respective associations with controllers, including order controllers, catalog controllers, and customer controllers. This consistency indicates that our approach is an accurate means of identifying application logic and interactions. However, there are some differences between the two models. The reverse-engineered model includes components not present in the gold standard, namely the catalog and customer stubs. These false positives indicate that our approach incorrectly identified these stubs or placeholder classes as components. The reverse-engineered model also includes these components in the list of false positives, suggesting that there may be a problem with redundant detection. The reverse-engineered model excludes several components and relationships in the gold standard.

In particular, the component and its dependencies, which act as an API gateway, and the service discovery component, which acts as a service registry, are conspicuous by their absence. In addition, the lack of a database and its dependencies is conspicuous. The exclusions above represent false negatives, demonstrating a limitation of our approach to identifying infrastructure-level components and their roles within the system architecture. It is important to note that the reverse-engineered model cannot identify certain relationships. In particular, the relationships between the customer client and the customer interface and between the catalog client and the catalog interface are absent. This indicates that although the components have been identified, their interactions with external services or interfaces have not been captured. This suggests deficiencies in the traceability of dependencies across service boundaries. The discrepancies can be divided into several categories based on the distinction between false positives and false negatives.

The first category of errors involves the inclusion of irrelevant components. This is illustrated by the stubs for the catalog stub and the customer stub. This suggests that our approach may have been overly inclusive or that the classification of components may have been incorrect. Duplicates in the false-positive list indicate the possibility of shortcomings in the filtering or aggregation mechanisms used during model generation. The second category of errors is the omission of infrastructure components. This under-identification suggests that our approach may be inaccurate when analyzing configuration files or external dependencies that define system-level services and middleware. The absence of the database also demonstrates a limitation in detecting backend services that may not be explicitly defined in the source code. The third category of errors relates to the inability to detect certain relationships between components and interfaces, particularly the lack of client-to-client and catalog-to-catalog connections. This incomplete linkage detection

challenges our approach to tracing service interactions, particularly in the context of client components that interface with external services.

Apache Web Server and Spring Boot Microservice Example In the case of the Apache web server and Spring Boot microservice examples, the reverse-engineered model can successfully identify the components of the software system, as evidenced by the successful mapping of the model to the actual software. These correspond to the shopping cart, product, and content services shown in the gold standard model. The figure provides a visual representation of each of these components. The internal components, including the shopping cart controller, shopping cart service, product controller, product service, and content controller, are accurately represented. In addition, the internal relationships between these components, particularly the dependencies where controllers require their respective services, are accurately described. However, there are apparent differences between the two models. One discrepancy is the absence of the web server component in the reverse-engineered model. In the gold standard model, this component is the central entity for routing requests to the various services through the shopping cart and product interfaces. This omission represents a false negative, as the reverse-engineered model fails to capture the request-routing mechanism implemented by the web server. The shopping cart and product interfaces, which should serve as entry points to the shopping cart and product services, are missing, further indicating deficiencies in the representation of external interfaces. In addition, the reverse-engineered model introduces false positives due to incorrect externalization of internal paths. To illustrate, the "count" and "add" paths are internal operations within the shopping cart service component; however, they are incorrectly represented as external interfaces. This misrepresentation suggests that these internal methods are externally accessible, which is not the gold standard model. The root interface is incorrectly associated with the product and shopping cart services. According to the gold standard model, the root interface is related to the content controller.

In summary, the reverse-engineered model has flaws. The first category of flaws concerns the absence of components and interfaces necessary to the system's functionality, particularly the web server component and the shopping cart and product interfaces. These are important to the delineation of the system's request-handling architecture. The second category of errors concerns the incorrect externalization of internal elements. In such cases, internal methods are misrepresented as external interfaces, thereby misrepresenting the accessibility and security boundaries of the system. The third category of errors is incorrect associations and relationships, including the erroneous association of interfaces with components that do not interact according to the gold standard model.

Movie Recommendation System A comparison of the gold standard model and the reverse-engineered model for the movie recommendation system reveals both similarities and differences in the representation of the software system's architecture. Identifying true positives shows that both models have comparable standard components and interface distributions. In particular, both models correctly identify the API and the recommendation client component, represented as the central controller in the reverse-engineered model.

The controller component and its relationships are accurately represented in both models. The models show a high degree of consistency in the presence of specific services. For example, both models include user, client, recommendation, and movie services. However, the terminology used in the reverse-engineered model is different, with these services referred to as user service, recommendation client service, and so on. Both models' user and movie interfaces are correctly associated with the user and controller. The internal relationships between the central, movie, user, and recommendation controllers are consistently represented in both models. However, an analysis of false positives and false negatives revealed several discrepancies. The false negatives highlight the absence of components and relationships in the reverse-engineered model that are present in the gold standard model. Note that specific components are missing from the reverse-engineered model. Note that the reverse-engineered model is missing the movie repository within the movie service and the user repository within the user service.

The recommendation service lacks two components: the user and movie repositories. In addition, the reverse-engineered model is missing several infrastructure components, including the service discovery and configuration services and their associated relationships. Notably, the model is missing meaningful relationships, including the recommendation controller's dependency on the user and movie repositories and the interactions between the recommendation client service and the user and recommendation interfaces. False positives indicate that the reverse-engineered model contains redundant or misrepresented components and interfaces. For example, the data associated with the movie dummy interface is redundant and has been incorporated into the movie interface. In addition, a new user interface was introduced as a discrete entity, integrating the user interface into the entity. The positioning of components such as the movie and recommendation services is incorrect and should be placed within the central controller rather than as separate entities. It is recommended that interfaces formatted as recommendations be consolidated under the heading of the recommendation interface, as they are redundant. The user service is incorrectly associated with the new user interface and inconsistent with the structure outlined in the gold standard model. These inconsistencies fall into three categories: the absence of components and relationships, the incorrect placement and naming of components, and redundant or misnamed interfaces. The absence of backend repositories, such as movie and user repositories, indicates that the approach may inaccurately capture data access layers. The lack of representation for infrastructure components such as service discovery and configuration services suggests that the approach does not adequately recognize and incorporate system-level services. The misplacement and misnaming of service components reflect the inherent difficulties in accurately mapping components to their corresponding controllers and services, which is challenging. The redundancies and inaccuracies in interface naming suggest deficiencies in the approach's ability to generalize and accurately identify interface patterns.

Spring Pet Clinic System A comparative analysis of the reference and reverse-engineered models of the spring pet clinic system reveals similarities and differences. Many components are correctly identified in both models. For example, the API gateway component is

represented as the API gateway controller in the gold standard model. It is analogous to the API gateway controller in the reverse-engineered model. Similarly, the visit resource and visit repository components are represented in both models as the visit service component, corresponding to the visit resource and visit repository components. The visit service component, including the visit resource and visit repository components, is represented precisely as the visit resource and visit repository components in the reverse-engineered model.

Similarly, the customer service component is represented consistently, with the pet and owner resources corresponding to the pet and owner resources, respectively. The associated repositories, the pet repository, and the owner repository are present in both models. The two models correctly map the relevant interfaces, such as owner and pet, to the visitor resources.

Similarly, mapping interfaces such as veterinarians, owners, pet types, and owners to their respective resources is accurate. However, there are discrepancies between the two models. The reverse-engineered model contains erroneous inclusions, such as the API gateway owners interface, which should be eliminated and replaced with the more accurate designation of API gateways, as seen in the gold standard model. In addition, false negatives have been identified where components such as the discovery service, config service, and admin server are present in the gold standard model but are missing from the reverse-engineered model and their associated relationships. The reverse-engineered model is deficient because it does not identify the dependencies of the API gateway controller on the customer service client and the visit service client. In particular, the interface owners associated with customer service are missing from the reverse-engineered model. These inconsistencies can be divided into categories of errors. The first category of mistakes is the absence of components and their associated relationships. To illustrate, the reverse-engineered model is missing the discovery service, the config service, and the admin server. The second category of errors relates to the lack of dependencies between components. This includes not recognizing the relationships between the API gateway controller and the customer and visitor service clients. The third category of errors relates to incorrect or incomplete representations of interfaces. This includes mislabeling API gateways and omitting owners as customer service interfaces. While the reverse-engineered model accurately captures several components and their relationships, it is deficient in its representation of specific components and dependencies missing from the gold-standard model. Discrepancies result from missing components, missing dependencies, and inaccurate representations of interfaces.

The Piggy Metrics System A comparison between the gold standard model and the reverse-engineered model reveals both similarities and differences. Both models correctly identify components and interfaces, including the notification recipient interface and its connection to the notification service component, represented as the notification service in the reverse-engineered model. Notably, several components, including the recipient controller, the email service imp, the recipient service, the account service client, and the recipient repository, are consistently present in both models. In the account service

component, the account controller, account service, authentication service client, statistics service client, and account repository are accurately represented in both models. In addition, the user and statistics interfaces are present, with the authentication service component corresponding to the user controller in the reverse-engineered model. The user service, user details service, and user repository are correctly identified in both models. The statistics service component is consistently represented, including its constituent elements: the statistics service, the statistics controller, the exchange rates service, the data point repository, the exchange rates client fallback, and the exchange rates client. The internal relationships of the statistics service are accurately represented in both models. However, inconsistencies in the form of false positives and false negatives were identified. The reverse-engineered model contains components not present in the gold standard model. These components include the frequency writer converter, the frequency reader converter, the data point reader converter, and the data point writer converter. The reverse-engineered model consists of a converter interface and associated components not included in the gold standard model. This discrepancy results in the generation of false positives.

Conversely, the gold standard model includes components and interfaces missing from the reverse-engineered model, resulting in false negatives. The reverse-engineered model cannot identify the gateway, configuration, and registry components and their associated relationships. In addition, the notification interface and its connection to the notification service are also missing, indicating that our approach may have failed to identify specific elements. These discrepancies can be grouped into error categories. The false positives are due to the inclusion of internal converter components and interfaces present in the source code but deemed redundant during the manual architectural analysis. This suggests that our approach may inadvertently prioritize low-level implementation details. The false negatives can be attributed to excluding high-level infrastructure components, such as the gateway, configuration, registry, and interfaces, including the notification interface. This suggests that the approach may have limitations in distinguishing specific architectural components, particularly those related to system configuration and external interfaces. In summary, the discrepancies between the two models can be attributed to the approach's tendency to include redundant low-level components and its inability to identify high-level architectural components and interfaces.

7.3.3.2. Project-Specific Validation

Retriever's approach to reverse engineering software architecture models integrates project-specific and technology-specific rules to increase model accuracy. Integrating two rules enables the exploration and representation of component implementations across projects. Model accuracy is evaluated based on precision, recall, and F_1 score metrics that assess the identification of components, connectors, interfaces, and external calls within the SEFF. The generation of new project-specific rules is quantified by the percentage of operators modified and the number of lines of code adjusted. The goal is to reduce these

Structural			Behavioral			Rule
Precision	Recall	F_1	Precision	Recall	F_1	LoC
`DescartesResearch/TeaStore`						
0.93	0.97	0.95	0.91	0.83	0.87	105
`ewolff/microservice`						
0.88	0.88	0.88	1.00	1.00	1.00	48

Table 7.4.: The results of a comparative analysis and the metrics used to evaluate the accuracy of the methodology used to infer structural and behavioral characteristics through project-specific extraction rules.

values to a minimum, possibly to zero, indicating that the existing technology-specific rules address the requirements of the new project without further modification.

In addition, the impetus for introducing new project-specific rules is to increase the accuracy of the architectural model. The accuracy of these integrations is measured by the number of new rules created and the corresponding lines of code involved. These metrics are not only based on the amount of effort involved, but also provide benchmarks for the efficiency of rule integration.

The new model improves, particularly in accurately identifying components, approaching the gold standard, and reducing false negatives. The model more accurately represents the interconnections and dependencies between components important to understanding system behavior. For example, improved interface and port detail both required and provided services, providing insight into inter-service communications, such as those facilitated by the registry client and service load balancer.

The project-specific rules' latest iteration addresses previous shortcomings by categorizing components according to their functional roles more accurately. In addition, improvements to the interface facilitate an examination of the relationships and interdependencies between components. These changes have improved the architectural representation, making the project-specific rules more accurate for software architecture analysis.

Table 7.4 shows that the Retriever approach can improve the accuracy of software architecture models by reducing the effort required to adapt new project-specific rules. These improvements strengthen the model's alignment with the gold standard, thereby increasing its accuracy relevance for architects and developers engaged in analyzing and understanding software architectures. These improvements to the Retriever approach facilitate reverse engineering of software architectures, promoting a detailed and accurate understanding of system structure and behavior.

Tea Store System A comparison between the gold standard and the newly improved reverse-engineered model shows an improvement in the accuracy of our approach. The new model can accurately identify the components of the gold standard, thereby reducing the number of false negatives. The new model exhibits high accuracy in correctly identifying components and several elements, including the authentication service, image service,

persistence service, recommendation service, registry service, and web user interface. The above elements are represented in the reverse-engineered model as authentication, image, persistence, recommendation, registry, and web user interface. The internal components of these services are accurately mapped and show a high degree of consistency with those in the gold standard. For example, the components are correctly positioned within the authentication service. In addition, the reverse-engineered model more accurately represents the relationships and dependencies between components. It includes interfaces and ports that symbolize the services required and provided, providing an understanding of the interactions between components. For example, the model depicts dependencies such as the registry client and the service load balancer for inter-service communication. Despite these improvements, some inconsistencies remain. The incorrect default interfaces remain, misrepresenting the actual interfaces of the system.

The reverse-engineered model fails to identify the database component and its associated relationships regarding false negatives. It is worth noting that certain data flow relationships are missing from the model. These include those between the web user interface and the registry service and between the registry service and the persistence service. These shortcomings indicate that although the identification of components has been refined, there is still room for further improvement in capturing communication between components. A comparison of the newly reverse-engineered model with the previous model reveals many areas where improvements have been made. As a result of the last model's misclassification of components into non-logical composites, the new model corrects this inaccuracy and correctly categorizes components according to their functional roles. The new model reduces the number of false positives by eliminating inappropriate component compositions and unnecessary engineering detail. This refinement results in a more refined and accurate representation of the system's architecture. The new model provides an improved interface representation that better represents interfaces and their connections to components.

However, some incorrect interfaces remain. This improved understanding allows a more detailed examination of the relationships and interdependencies between components. The improved project-specific rules have proven accurate in identifying previously undetected services, reducing the number of false negatives. Including components brings the reverse-engineered model closer to the gold standard. Incorporating inter-component dependencies, as evidenced by the improved capture of dependencies and required interfaces, such as those involving the registry client and service load balancer, represents an improvement in reverse engineering. This reflects an understanding of the system's internal mechanisms and the nature of its component interactions. In general, the recently improved project-specific rules show progress in accurately representing the architectural configuration of the system. While some errors remain, particularly in incorporating technical implementation details and incomplete data flow representations, the improvements introduced by the project-specific rules have improved the model's alignment with the reference architecture. Reducing false positives and negatives indicates a more accurate approach, providing valuable insights for software architecture analysis.

Microservice Sample The gold standard and the newly improved reverse-engineered model show agreement in identifying the components and their interactions within the software system. Both models accurately represent the order, customer, and catalog services, including their internal components. These components include the order controller, order service, customer client, catalog client, order repository, customer application, customer repository, catalog application, catalog controller, and item repository. The internal relationships between these components are represented consistently, reflecting the dependencies and interactions necessary for the system's functionality. The models illustrate interfaces related to orders, catalogs, and customers. In addition, the relationships between these interfaces and their corresponding controllers are accurately represented. The model's precision captures the dependencies of the order service on the customer and catalog clients, as well as the associations between controllers and repositories in the customer and catalog components. This consistency demonstrates that our approach has accurately identified the architectural elements and their interactions. Despite these similarities, there are differences between the gold standard and the newly reverse-engineered model. The reverse-engineered model includes components not present in the gold standard, namely the catalog and customer stubs. It can be postulated that these components represent false positives, indicating that our approach mistakenly identified stubs or placeholder classes as components within the architecture.

The reverse-engineered model excludes specific components and relationships in the gold standard. In particular, the reverse-engineered model omits the API gateway, associated dependencies, and the service registry component. These exclusions represent false negatives and indicate a limitation in our approach to identifying infrastructure-level components and middleware services integral to the overall system architecture. A comparison of the recently reverse-engineered model with the previously reverse-engineered model shows an improvement in the accuracy of the rules employed. The previous model identified additional false positives, including duplicates of the catalog stub and customer stub components. The new model eliminated these redundancies, reducing the number of false positives and improving the accuracy of component identification. The previous model produced false negatives regarding the relationships between the customer client and the customer interface and between the catalog client and the catalog interface. The new model corrected these shortcomings by accurately capturing these relationships. The new model's customer and catalog client components now provide customer and catalog interfaces that reflect previously unrecognized dependencies and interactions. This improvement illustrates our approach's ability to capture inter-service communication and external dependencies. The previous model's shortcomings, which did not include the database component and its associated relationships, have been addressed in the new model, which now consists of the necessary associations with the repositories. Notwithstanding the continued absence of the API gateway and service registry components, the reduction in false negatives and the inclusion of previously missing relationships illustrate an improvement in the accuracy of our approach.

In summary, the recently extended project-specific rules show improved concordance with the gold standard and accurately identify components and their interactions. In addition, they reduce the number of false positives by eliminating redundant or unimportant com-

Vulnerability		
Precision	Recall	F_1
ewolff/microservice		
0.98	0.87	0.92
georgwittberger/apache-spring-boot-microservice-example		
1.00	0.81	0.89
spring-petclinic/spring-petclinic-microservices		
1.00	0.60	0.75
sqshq/piggymetrics		
0.84	0.99	0.91

Table 7.5.: The comparison results and the metrics for the accuracy of the approach in generating vulnerability properties are presented.

ponents and correcting previous omissions of relationships between client components and interfaces. These developments demonstrate that incorporating project-specific rules into our approach has improved its ability to model system architecture with greater precision.

7.3.3.3. Vulnerability Validation

In the preliminary phase, a PCM was extracted from the publicly available artifacts within the end-to-end case study code repository. The PCM offered a structural and behavioral view of the case study system. Subsequently, an automated process was conducted to annotate the architectural model components with the identified vulnerabilities. The vulnerability annotation process was performed utilizing the third-party tool Snyk CLI and the NVD, thus eliminating the necessity for manual intervention.

The subsequent phase assessed the accuracy of the vulnerability annotations for each case study. The results will provide insight into the efficacy of our approach to identifying vulnerable components. This entailed constructing attack propagation models that were both syntactically and semantically valid.

Some vulnerability annotations were duplicated during the annotation process. Nevertheless, these duplications did not impact the subsequent analysis. To guarantee the precision and dependability of the analysis, identical instances were excluded if the data they contained was identical. In such cases, the vulnerability and the affected component were treated as a single entity for analysis.

Table 7.5 presents the findings of the vulnerability accuracy validation. The initial column of this table enumerates the case studies, followed by the respective values for precision, recall, and F_1 score for each case study. Precision is the proportion of correctly identified vulnerabilities out of the total identified by the annotation process. Recall is defined as the proportion of correctly identified vulnerabilities out of the total number of vulnerabilities present in the system. The F_1 score, which is the harmonic mean of precision and

recall, measures the accuracy of the annotation process. The application of these metrics permitted a detailed evaluation of the precision and recall of the automated vulnerability annotation process.

Table 7.5 presents the results of the accuracy evaluation of our approach for four case studies: the microservice example, the movie recommendation system, the Spring Pet Clinic System, and the pig metrics system. The evaluation of each case study is based on the number of true positives, false positives, false negatives, precision, recall, and F_1 score. The lowest precision observed is 0.84 for the Pig Metrics system, which, despite having the highest number of vulnerability types, indicates an overestimation by our approach. This overestimation occurs because the approach indiscriminately assigns vulnerabilities in the parent project to all child projects, leading to the incorrect annotation of some unaffected child projects. This results in an increased number of false positives, which reduces the results' accuracy. It is of utmost importance that this issue be addressed in future work to improve the specificity of our vulnerability annotations.

In contrast, the lowest recall observed was 0.60 for the Spring Pet Clinic System case study. This low recall can be attributed to Snyk not annotating all the CVEs identified by Snyk as part of the analysis. In particular, when multiple CVEs are associated with a single vulnerability, the current approach only annotates one, resulting in missed vulnerabilities and an increased prevalence of false negatives. Refining the approach to include all specified CVEs is important to improving recall in future analyses.

The Movie Recommendation System case study provides an example of a scenario where precision and recall achieve a perfect score of 1.00, indicating that all vulnerabilities have been accurately annotated with no false positives or negatives. This results in an F_1 score of 1.00, representing a balance between precision and recall. The Microservice Sample case study showed a high level of precision 0.98 and a relatively low level of recall 0.87, resulting in an F_1 score of 0.92. The slight decrease in recall indicates that some vulnerabilities may have been missed, and further investigation into the underlying causes is required.

In the Piggy Metrics System case study, while the recall is high at 0.99, the precision is comparatively lower at 0.84 due to the abovementioned overestimation problem. The high recall indicates that the proposed approach accurately identifies the most current vulnerability, an aspect of security validations. However, this is achieved at the cost of reduced precision due to increased false positives.

The results indicate that while our approach accurately detects vulnerabilities, as evidenced by high recall values, it tends to overestimate vulnerabilities, affecting precision. In the context of security analysis, this overestimation may be an acceptable trade-off, as the goal is to identify all plausible security issues, even at the expense of additional effort to address false positives. Nevertheless, further work is needed to refine the approach to reduce false positives without compromising recall, which would improve its applicability. Further work should focus on increasing the specificity of vulnerability assignment to child projects and ensuring that all relevant CVEs are flagged, thereby optimizing precision and recall.

8. Discussion and Validity

This section examines the implications of our findings and assesses the validity of our research approach and results. Our experimental evaluation employs the GQM approach, which systematically defines objectives, question formulations, and metrics identification to evaluate our Retriever approach.

First, we discuss the applicability of our approach in different technology ecosystems. We evaluate the ability of our Retriever approach to generate architectural models, as demonstrated by empirical validations (Section 8.1.1). Finally, we then evaluate the accuracy of the generated models to determine their accuracy in representing the underlying software systems' structural, behavioral, and security-related properties (Section 8.1.2).

An aspect of our discussion is the validity of our results. We divide this into four areas: internal validity, focusing on the consistency of our experimental results; external validity, discussing the generalizability of our approach; construct validity, ensuring that our measures accurately reflect the theoretical constructs they are intended to assess; and reliability, emphasizing the reproducibility of our results across different settings (Section 8.2). This validity examination intends to demonstrate the quality of our research and the reliability of our conclusions.

8.1. Answering Our Evaluation Questions

We used the GQM approach to evaluate our Retriever approach in our experimental evaluation. This structured approach defines specific goals, formulates relevant questions, and identifies precise metrics of empirical assessment. First, the two goals for our approach were defined, focusing on its applicability in different environments (Section 8.1.1) and the accuracy of the architectural models it generates (Section 8.1.2). Questions were then developed to evaluate the capabilities of our approach, particularly regarding its functional and non-functional attributes, such as scalability, precision, and recall. Each question is discussed in the following section, along with the corresponding answers from the experimental evaluation conducted.

8.1.1. Applicability to Relevant Software Systems

Our reverse engineering approach's applicability to relevant software systems was demonstrated through two dimensions, each addressing different aspects of the system's architecture and supporting validation.

In Section 8.1.1.1, the relevance of the systems was confirmed through an evaluation employing GitHub metrics, including commits, contributors, forks, and stars. This analysis confirmed that the selected systems were widely used and represented real-world applications with technological diversity and complexity. The applicability of our approach was validated by incorporating systems with a range of characteristics, including those with large code bases and multiple technologies, such as Java and Docker, as well as smaller, less complex systems. The application of our approach in these scenarios demonstrated its capacity to accommodate a range of architectural styles and technological environments, thereby reinforcing its practical relevance.

In Section 8.1.1.2, the scalability and portability of our approach were validated through the demonstration of consistent performance across several real-world systems. The approach was tested for scalability by measuring the execution times as a function of the size and complexity of the input variables, including the number of LoC and the number of components. The results of multiple case studies, including complex systems, demonstrated minimal variation in execution times and confirmed practical performance as system complexity increased. Our approach's practical scalability across diverse environments supports its applicability to real-world applications.

In Section 8.1.1.3, the generation of valid architectural models abstracted from the source code was confirmed through strict adherence to syntactic and semantic standards. The experimental evaluation demonstrated that no violations of OCL constraints occurred, and the generated architectural models exhibited reduced complexity, abstracting elements from large datasets. These models adhere to defined syntactic and semantic correctness and support handling complex software systems.

In conclusion, evaluating these two dimensions provides evidence of our approach's applicability to various relevant software systems. The approach meets the criteria for generating valid architectural models, indicating its practical scalability in real-world settings.

8.1.1.1. To What Extent Are the Systems Relevant, Regarding Their Use by the Community, Their Size, and Their Use of Heterogeneous Technology?

The experimental evaluation of our approach to reverse engineering software architectures was conducted by our GQM plan, which sought to confirm the applicability of our approach to relevant software systems. This was evaluated regarding the relevance and technological heterogeneity of the end-to-end case studies, as defined by the metrics employed.

The initial section of our discussion is dedicated to examining the relevance of the end-to-end case studies, with particular attention paid to evaluating their significance based on GitHub metrics. This includes the number of commits, contributors, forks, and stars. These metrics show applicability in identifying widely used, actively maintained, and widely forked systems, indicating their importance and widespread use within the developer community. This selection criterion supports that our evaluation is based on systems that are both relevant and representative of real-world applications, thereby enhancing the practical applicability of our findings.

Concurrently, an evaluation of the technological heterogeneity of these end-to-end case studies was conducted. An evaluation of each system's technological diversity was conducted by analyzing the number of LoC and source files and an evaluation of various technologies used. The selected systems employ multiple technologies, including Java with the Spring Framework for back-end services, ECMAScript for front-end development, and Docker for containerization. These observations are consistent with the structural characteristics commonly noticed in real-world applications. The variety of technologies employed underlines the applicability of our approach to different architectural styles and frameworks.

The efficacy of our approach was indicated through a series of case studies of selected end-to-end scenarios. An analysis was conducted on each system, with metrics tabulated based on its popularity on GitHub (as indicated by the number of stars, forks, and commits) and technical complexity (quantified by the number of files and the total LoC). The structured analysis allowed for a comparative understanding of each system and demonstrated the scalability of the approach across systems of varying size and technological composition.

In conclusion, the evaluation's results suggest that the selected metrics are relevant to the software systems analyzed. This offers affirmative support for the evaluation question regarding the systems' relevance and technological heterogeneity, thereby supporting the evaluation of our approach's scalability to relevant real-world applications.

8.1.1.2. To What Extent Is the Approach Scalable and Transferable to Relevant Real Systems?

The results of evaluating the scalability of our approach to real-world systems support the assumption that our approach scales practically with the input variable. The results illustrate the viability of implementing our approach in many real-world scenarios. In each end-to-end case study, the GitHub Action [Git24] was executed ten times independently with consistent inputs, revealing minimal variation in execution times with a maximum deviation of only one second. This consistency serves to reinforce the applicability of the automated process that has been built into our approach.

A comparative analysis of the execution times of the GitHub Actions and local test runs has demonstrated that our approach exhibits consistent performance across different environments, thereby indicating its applicability. Graphical representations that correlate execution time with various input and output sizes provide insight into the scalability

of the process—in particular, the execution time exhibited a direct correlation with both the LoC and the number of artifacts. It was observed that longer execution times were required for larger code bases and complex artifact configurations. This trend was observed consistently across all metrics, indicating scalable performance.

The correlation between execution time and the number of components in the final architectural models provides additional validation. Models comprising more components required a longer execution time, which reflects the increased processing necessary to map more complex architectures accurately. This correlation between execution time and output complexity further indicates the approach's scalability.

The specific case studies illustrate this scalability, providing evidence of the approach's applicability. To illustrate, the `corona-warn-app/cwa-server` case study, which had a larger input size and employed many technologies, required the longest execution time. In contrast, the `piomin/sample-spring-oauth2-microservices` case study, which had smaller components and a smaller code base, exhibited the shortest execution time. These observations align with the anticipated outcomes, whereby more significant inputs result in longer execution times. An analysis of the remaining case studies between the two extremes reveals an increase. This observation lends support to the assumption that our approach is practically scalable.

In conclusion, the analysis and the obtained results provide evidence to support the assertion that our automated reverse engineering approach scales practically with input size and adaptively responds to the complexity of the output models. This demonstrates the approach's applicability to actual systems, providing a foundation for its application in complex software environments. The consistent execution times for similar input or output sizes offer further evidence of the practical scalability of our approach, thus making it a valuable tool in the field of software architecture modeling.

8.1.1.3. Does the Approach Lead to Valid Instances of an ADL That Abstract from the Source Code?

The integrated analysis of the first and second metrics from our experimental evaluation provides evidence supporting the efficacy of our approach to generating architectural models that abstract from source code while adhering to syntactic and semantic standards. The first metric, which pertains to the number of OCL constraints violated in the reverse-engineered models, confirms that no such violations occurred. This was achieved by integrating our approach into a continuous integration and continuous delivery pipeline that enables the automated generation and validation of UML and PCM models. Using PlantUML for syntactic representation and the EMF Validation Framework for supplementary syntactic and semantic examinations ensures that all models comply with the gold standards and validation rules. The absence of errors or warnings in the Eclipse environment, where the PCM models were validated, provides further evidence of the syntactic and semantic correctness of the models.

The second metric, which analyzes the ratio of source files and total LoC to the number of components, also provides evidence supporting the applicability of our approach in managing and simplifying complex software systems. The empirical evidence demonstrating a slower rate of increase in the number of components relative to the rise in the code base's size illustrates our approach's efficacy in abstracting and reducing complexity. This is particularly evident in case studies such as `corona-warn-app/cwa-server`, where an input of over 850 source files and 34 000 LoC was abstracted into just ten components in approximately nine seconds.

These examples demonstrate the approach's applicability, which can produce meaningful and manageable architectural models from large and complex software systems. The reduction in complexity, as evidenced by the high reduction percentages, illustrates our reverse engineering approach's abstraction capability. This capacity to transform larger software systems into abstracted architectural models exemplifies the practical applicability of our approach to understanding and maintaining software architectures.

In conclusion, the results of both metrics provide evidence that our approach meets the criteria for generating valid architectural models. The generated models reflect and simplify the original software systems, demonstrating the approach's applicability in abstracting from source code. The dual validation on syntactic and semantic dimensions and the ability to handle complexity at different scales indicate that our reverse engineering approach is practically applicable. This means the architectural models can be relied upon for subsequent analysis and decision-making. This positive result answers the central validation question, demonstrating that our approach creates valid architectural models that abstract from diverse software artifacts.

8.1.2. Accuracy in View Generation

The accuracy of our reverse engineering approach was evaluated using four research questions focused on identifying structural and behavioral properties in software architectures and transferring vulnerabilities from security analyses to the vulnerability model. The evaluation aimed to determine the extent to which our retriever approach accurately identified these properties and vulnerabilities. In addition, the evaluation assessed the effort required to define precise project-specific rules to improve model accuracy. The results of the evaluation are discussed in the following section.

Section 8.1.2.1 showed promising accuracy in identifying structural features, as evidenced by applying quantitative metrics such as precision, recall, and the F_1 score. Despite high precision, certain limitations were evident, including misidentification and omission of components in different systems. The results suggest the potential of tailored extraction rules to increase the accuracy of automated reverse engineering processes.

Section 8.1.2.2 focused on the accuracy of identifying behavioral and dependency properties in reverse-engineered software architectures. Initial results showed variability in

accuracy; in particular, our approach struggled to correctly capture meaningful behavioral interactions and dependencies, where interactions and data flows were inadequately represented.

Section 8.1.2.3 highlights the accuracy of identifying system vulnerabilities. Using the Snyk CLI and the NVD, this process demonstrated high precision and recall across different systems. However, variations in accuracy were evident. These discrepancies were due to overgeneralization of vulnerability assignments and incomplete CVE annotations, resulting in overestimation and missed vulnerabilities. Despite these issues, the high recall values indicate the accuracy of the approach in capturing most potential security issues. However, the false positives indicate the need to refine the annotation process to improve accuracy without sacrificing vulnerability detection.

Section 8.1.2.4 demonstrated the value of these custom rules in improving model accuracy. The evaluation compared the accuracy of models that used only technology-specific rules to those that improved with project-specific rules. This investment in coding improved model accuracy and alignment with the gold standards. The results indicate the need for project-specific customization to capture the system architecture, thereby improving architectural models' utility in real-world scenarios.

8.1.2.1. Have the structural properties of the architecture been correctly identified?

The objective was to evaluate the accuracy of our Retriever approach in reverse engineering software architectures, focusing on determining the extent to which the structural features of the architectures were accurately identified. To this end, we conducted experimental validations using seven gold-standard case studies, including the Acme Air System, the Tea Store System, and the Spring Pet Clinic System. These case studies were analyzed, creating manual reference models that served as benchmarks for evaluating the models generated by our Retriever approach.

The evaluation process involved applying technology-specific extraction rules to each case study and comparing the resulting architectural models with the manually constructed gold standards. Quantitative metrics, including precision, recall, and the F_1 score, were used to assess the accuracy of the identified structural properties. The results indicate that our Retriever approach exhibited high precision and recall in some instances. Precision values ranged from 0.78 to 1.00 for structural properties, while recall values ranged from 0.64 to 0.91 across the different case studies.

Despite these encouraging results, the validation revealed some limitations. Our approach occasionally misidentified components, resulting in false positives, and sometimes failed to identify components, resulting in false negatives. For example, the reverse-engineered model incorrectly identified certain elements as standalone components within the Acme Air system and did not include components such as the web application and data service. Similar problems were observed in other case studies, including the Tea Store system, the Apache web server, and the Spring Boot Microservice Example, where components and their relationships were either misrepresented or not captured.

In conclusion, the evaluation demonstrated that our Retriever approach accurately identifies the structural properties of software architectures. In most cases, our approach accurately captured the core components and their interactions. However, specific errors affected the overall accuracy of the generated models. To address these shortcomings, project-specific extraction rules were introduced in Section 8.1.2.4 to improve model accuracy. This allowed the architectural components and their relationships to be better identified and improved the accuracy of automated reverse engineering. Future work should refine the extraction rules and address the identified limitations to achieve even higher precision and recall in identifying structural properties.

8.1.2.2. Has the behavior of the component-based software architecture with dependency properties been correctly identified?

The evaluation aimed to evaluate the accuracy of our Retriever approach in reverse engineering software architectures, particularly emphasizing the accuracy of the identified behavioral and dependency properties. To this end, we conducted experimental validations using a set of eight gold-standard case studies, including systems such as the Acme Air System, the Tea Store System, and the Microservice Sample System. These case studies were subjected to analysis to create manual reference models that served as benchmarks for evaluating the models generated by our Retriever approach.

The evaluation process involved applying technology-specific extraction rules to each case study and comparing the resulting architectural models with the manually constructed gold standards. Quantitative metrics, including precision, recall, and the F_1 score, were used to assess the accuracy of the identified behavioral and dependency properties. The results indicate that while our Retriever approach demonstrated proficiency in identifying structural properties, its accuracy in capturing behavioral properties was generally inferior. For behavioral properties, precision values ranged from 0.63 to 1.00, while recall values ranged from 0.30 to 0.85 across the case studies.

The analysis indicated that our approach occasionally misrepresented or failed to capture behavioral interactions and dependencies between components. For example, in the case of the Acme Air system, the reverse-engineered model did not include several components and their interactions, such as the data service component and its connections to other services. Similarly, in the Tea Store system, our approach did not accurately represent data flows and dependencies between services, such as the registry service and other components. This resulted in a behavioral recall of only 0.30.

In summary, there are limitations in accurately capturing behavioral and dependency aspects using only technology-specific rules. To address these shortcomings, in Section 8.1.2.4 we introduced project-specific extraction rules to improve the accuracy of behavior and dependency identification. Incorporating these rules resulted in improvements, as shown by the increased precision and recall values. This refinement allowed for exploring relationships and interdependencies between components, resulting in models more closely aligned with gold standards. It is recommended that future work refine the extraction

rules and address the identified limitations to achieve even higher precision and recall in determining behavioral and dependency properties within complex software systems.

8.1.2.3. How accurately can we transfer vulnerabilities from static security analysis to the architectural vulnerability model?

We evaluated the accuracy of transferring vulnerabilities from security analyses to a vulnerability model within software architecture representations. Using the Snyk CLI and the NVD, we used an automated process to annotate architectural model components with identified vulnerabilities. This approach aimed to eliminate the need for manual intervention and reflect the realities of real-world development environments by integrating the vulnerability annotation process into a CI/CD pipeline.

To evaluate the accuracy of this approach, we conducted experiments on four gold-standard case studies: the Microservice Sample System, the Apache Spring Boot Microservice Example, the Spring Pet Clinic System, and the Piggy Metrics System. The accuracy of vulnerability annotations was evaluated using quantitative metrics. Precision was defined as the proportion of correctly identified vulnerabilities out of the total number determined by the annotation process. Simultaneously, the recall was calculated as the proportion of correctly identified vulnerabilities out of the system's newly identified vulnerabilities. The F_1 score, defined as the harmonic mean of precision and recall, was used to evaluate the overall accuracy of the annotation.

As shown, the results indicate different levels of precision and recall across the case studies. The Apache Spring Boot Microservice Example demonstrated high accuracy with a score of 0.98 and a recall of 0.87, resulting in an F_1 score of 0.92. The Microservice Example had a precision of 1.00, while its recall was 0.81, resulting in an F_1 score of 0.89. The Spring Pet Clinic system also had perfect precision but a lower recall, resulting in an F_1 score of 0.75. The Piggy Metrics system had a precision of 0.84 and a high recall of 0.99, resulting in an F_1 score of 0.91.

The discrepancies in precision and recall were attributed to specific limitations of our approach. The reduced precision observed in the Piggy Metrics system can be attributed to the indiscriminate assignment of vulnerabilities from parent projects to all child projects. As a result of this approach, false positives were identified and vulnerabilities were overestimated. In contrast, the lower recall observed in the Spring Pet Clinic system was due to the annotation process not including all CVEs associated with a single vulnerability. As a result, vulnerabilities were excluded, and the number of false negatives increased.

Despite these limitations, the high recall values observed in the case studies indicate that our approach accurately identified most system vulnerabilities. This is particularly important in security analysis, where the goal is to identify as many potential security problems as possible. However, false positives, as evidenced by lower accuracy in some cases, suggest that our approach may overestimate vulnerabilities, potentially leading to unnecessary remediation efforts.

In conclusion, the evaluation demonstrated that vulnerabilities are accurately transferred from security analyses to the vulnerability model, as evidenced by high recall rates, supporting vulnerability detection. However, the level of accuracy observed in the case studies was variable due to the over-assignment of vulnerabilities and the incomplete nature of CVE annotations. The results suggest that while our approach is viable for vulnerability detection, further refinements were made to reduce false positives without compromising high recall. Further research should focus on increasing the specificity of vulnerability assignments to improve accuracy and achieve a balance between detecting all plausible security issues and minimizing unnecessary remediation efforts.

8.1.2.4. How much effort is required to define more accurate project-specific rules?

The evaluation examined the effort required to define more precise project-specific rules in the context of reverse engineering software architectures using our Retriever approach. To address this issue, we performed project-specific validation to assess the accuracy of the rules. We focused on evaluating the accuracy of architectural models after integrating new project-specific rules and comparing the results to those obtained using only technology-specific rules.

The metric to quantify the effort was the number of newly formulated project-specific rules and the total LoC defining them. In an ideal scenario where existing technology-specific rules are sufficient, the need for additional project-specific rules would be minimal, approaching zero. However, the evaluation indicated that the integration of project-specific knowledge required some level of additional rule definition.

We presented a comparative analysis and the metrics used to evaluate the accuracy improvements achieved by project-specific rules for two case studies: the Tea Store System and the Microservice Sample System. In the case of the Tea Store System, incorporating project-specific rules resulted in adding 105 LoC, resulting in structural precision and recall scores of 0.93 and 0.97, respectively, as well as behavioral precision and recall scores of 0.91 and 0.83. Similarly, 48 LoC were added for the Microservice Sample System, resulting in perfect behavioral precision and recall scores of 1.00 and structural precision and recall scores of 0.88.

These results indicate that a small investment in additional LoC is sufficient to define more accurate project-specific rules. Incorporating these rules increased the accuracy of the reverse-engineered models. In particular, there was a reduction in false positives and negatives, bringing the models closer to gold standards. These results illustrate the practical value of investing effort in customizing rules for specific projects, as evidenced by the improved accuracy.

The evaluation also showed that, while technology-specific rules provide a foundation, they may not fully capture the complexities of individual projects. Project-specific rules are essential to addressing architectural patterns and component interactions that cannot be generalized across systems. By formulating and integrating these custom rules, other

researchers can more accurately identify components and their interdependencies, thereby increasing the accuracy of architectural models.

The effort required to define more accurate project-specific rules is quantifiable and manageable. In the cases studied, the number of LoC ranged from 48 to just 105. This investment yields benefits in terms of improved model accuracy. The results indicate that while technology-specific rules are beneficial, supplementing them with project-specific rules is essential to achieving higher accuracy in architectural reverse engineering. This approach increases the practical relevance of the models for architects and developers engaged in analyzing and understanding complex software architectures.

8.2. Threats to Validity

In the context of scientific research, particularly in the case of study-based evaluations, it is essential to address potential threats to the validity of the findings to support the credibility of the results. In the context of scientific research, the term *validity* is used to describe the extent to which the findings of a case study-based evaluation can be appropriately generalized, and the evaluation employed can be considered an accurate reflection of the phenomena under study. The guidelines proposed by Runeson et al. [RH08; Run+12] provide a structured framework for evaluating these aspects, supporting that a case study's findings are reliable across contexts.

Internal Validity refers to the extent to which the findings of a case study-based evaluation can be attributed to the variables that the researchers have studied rather than to external influences. The objective of supporting internal validity is to demonstrate that the evaluation process has satisfactorily accounted for potential confounding variables and that the results accurately reflect the intended outcomes of the evaluation design.

External Validity refers to the extent to which the findings of a case study can be generalized beyond the immediate context of the case study itself. This necessitates an evaluation of the extent to which the conclusions derived from a specific context can be transferred to other settings that may exhibit differences in key characteristics, such as technology, geography, or temporal factors.

Construct Validity pertains to the suitability of the evaluation instruments and procedures utilized in the case study-based evaluation. This aspect of validity relates to the extent to which the evaluation and measures are employed in an evaluation to evaluate the theoretical constructs they intend to consider.

Reliability pertains to the consistency of the measurement process and the reproducibility of the case study-based evaluation's results. A reliable evaluation is one in which other researchers can conduct the same experiments or analyses under the same conditions and obtain comparable results.

By addressing these four aspects of validity and following the guidelines set forth by Runeson et al. [RH08; Run+12], researchers can improve the strength of their case studies, supporting both the precision of their measurements and the integrity of their conclusions. This approach enhances the credibility of the findings and reinforces the research's contribution to the scientific discourse.

8.2.1. Internal Validity

Internal validity is an aspect of any scientific investigation, as it supports that the observed results are attributable to the variables under study and that the research evaluation process is sound. It supports the claim that the causal relationships identified by the research are genuine and can be attributed to the variables under study. In the context of case study research, it is essential to demonstrate that the results are not influenced by external factors or biases inherent in the evaluation design, particularly when evaluating software systems and our Retriever approach. This subsection will examine the internal validity of the case study-based evaluation through three focused discussions: objectives and evaluation processes, challenges and limitations, and standardization and consistency. These discussions aim to examine the internal validity of our research, supporting the conclusion that it is both sound and defensible within the scientific community.

8.2.1.1. Evaluation Objective and Processes

In addressing threats to validity within our experimental evaluation, particular attention is paid to the internal validity of the objective and techniques of our experimental evaluation. This evaluation aims to describe the architecture of each gold standard system accurately. This is accomplished by examining the source code, configuration files, and expert insights. The goal is to ensure that only the intended components, interfaces, roles, and associated data types are correctly identified. It is essential to ensure that the results of automated reverse engineering accurately reflect the underlying software architecture and are functional. Achieving this goal requires comparing the reverse-engineered architectural models and the gold standard model. This comparison is not just a verification step, but a component of our goal and process to align the automated results with the validated architectural model. Discrepancies between the reconstructed models and the existing gold standard can provide insight into potential areas for improvement in our reverse engineering approach.

8.2.1.2. Challenges and Limitations

As part of our experimental evaluation, we focus on addressing the challenges and limitations inherent in our evaluation process, particularly regarding internal validity. This evaluation centers on identifying components and their interfaces, representing a singular view of the system. This single view may result in discrepancies between the reconstructed

model and the actual architecture, particularly if the underlying documentation that guides the reconstruction process is outdated or incomplete. In the absence of documentation, there is a risk of an inaccurate representation of the derived architecture, which could misrepresent the system's actual configuration.

Our experimental evaluation's integrity is vulnerable to inconsistencies due to the subjective nature of manually constructing gold standards. These are essential for validating automated reverse engineering results and depend on the expertise and interpretation of the individuals responsible for creating them. The technical environment in which reverse engineering approaches can be used can affect the results. For example, using platforms such as Eclipse, which can exhibit different behaviors in different configurations, introduces an additional layer of complexity. Such environmental variability can potentially influence the behavior of the approaches used for model generation, which can impact the generated models' reliability. It is essential to control and standardize the software environment to mitigate the impact of this variability on the results.

To address these challenges, it is necessary to refine documentation review processes to support completeness and currency, standardize the development of gold standards to minimize subjectivity, and control the software environment to reduce the influence of external variables. These steps are essential for enhancing the reliability of our Retriever approach and ensuring that the reconstructed architectures accurately represent the systems under study.

8.2.1.3. Standardization and Consistency

The internal validity of our research depends on the standardization and consistency of the reverse engineering process to address potential variability and support the reliability of the results. The open implementation of our Retriever approach in a CI/CD pipeline aims to standardize the experimental evaluation for automated reverse engineering, thereby reducing subjectivity and increasing consistency in the interpretation of data across different case studies.

An aspect of our Retriever approach is using a consistent configuration of the Eclipse platform. The software architecture models generated must adhere to a standardized process to support their reliability and that comparisons across analyses yield valid insights. By maintaining a consistent approach to configuration, discrepancies caused by variations in the software environment are mitigated, thus supporting the reproducibility and comparability of results.

Implementing these measures – consistent approach configuration and comparison to gold standards – is intended to improve the internal validity of the research. Such standardization is of consequence, as it supports the idea that our case studies' outcomes are consistent across analyses and accurately reflect the underlying architectures intended to represent. This experimental evaluation reinforces the overall reliability of our findings in the automated reverse engineering of software architecture models.

8.2.2. External Validity

External validity describes the extent to which findings derived from a case study can be applied beyond the specific scenarios under study. This aspect of research validity is critical when applying findings from particular cases to a context. This is particularly challenging in case study research because each case is complex. This subsection is divided into generalizability concerns, case study selection with diversity, and future directions to address external validity. In conclusion, these discussions evaluate the extent to which the findings from our case study-based evaluation can be expected to hold in different contexts. This supports the relevance of our research to the communities.

8.2.2.1. Generalizability Concerns

The issue of generalizability emerges as a concern when evaluating the external validity of our experimental evaluation. The possibility of extrapolating the findings from our case studies to more encompassing contexts is constrained, particularly when extending these findings to systems that employ diverse technologies or architectural styles. These limitations result from the systems selected for evaluation – those operating in the web services domain.

The evaluation was concerned with systems developed using object-oriented and structured programming paradigms. While this focus allows for a complete understanding of these paradigms, it also raises questions about the generalizability of our findings to systems based on alternative programming paradigms, such as functional or logic programming. These systems may be distinct in their design and operational dynamics. The distinctive attributes of these paradigms, which can impact architectural structuring and component interaction, may not be captured by our Retriever approach optimized for object-oriented and structured environments.

The restricted range of technologies and architectural styles investigated in the case studies may not adequately represent the diversity and intricacy of modern software systems, which frequently integrate multiple programming paradigms and hybrid architectural styles. This specificity may restrict the extrapolation of our results to other types of systems not included in the experimental evaluation, which could limit the applicability of our reverse engineering approach.

To address these concerns regarding generalizability, future research could expand the range of technologies and architectural styles examined to include diverse programming paradigms and complex system configurations. Such an expansion would evaluate our Retriever approach's applicability and accuracy. Understanding their applicability across different technological contexts would also increase the results' external validity.

8.2.2.2. Case Study Selection and Diversity

Selecting and diversifying the case studies is essential when evaluating our experimental design's external validity. Our evaluation process employs predominantly external case studies to mitigate the risk of overfitting and guarantee that the results are not unduly influenced by the peculiarities of internal systems that may not be representative of industry practices. However, the inherent limitation in the number of selected case studies may challenge the generalizability of our findings.

Although the case studies were selected to encompass a range of heterogeneous systems, their diversity is anchored in a specific technological spectrum, with a predominant focus on prevalent languages such as Java or ECMAScript. Although this focus is advantageous for a complete evaluation within this technological context, it may restrict the breadth of our findings. While exhibiting application and diversity, the systems under consideration represent only a subset of the technologies.

Considering this limitation, a concerted effort was made to extend the range of technologies included in the studies. The objective was to incorporate a spectrum of programming languages and architectural frameworks, thereby enhancing the diversity of the case studies and, consequently, the applicability of our findings across diverse technology domains. The inclusion of multiple technologies is intended to enable analysis and to evaluate the applicability of our reverse engineering approach when applied to diverse software architectures and development environments.

Such a diversified selection approach is critical to validating the reverse engineering techniques' applicability and supporting that the conclusions drawn from the experimental evaluation apply to a range of real-world scenarios. Subsequent iterations of this research may benefit from an even more expansive selection of case studies, potentially including more technologies, to evaluate the generalizability of the reverse engineering approach in diverse technological contexts. Such an expansion would assist in overcoming the limitations of the existing selection and validate our Retriever approach's applicability across the technological spectrum.

8.2.2.3. Future Directions

In considering potential avenues for enhancing the external validity of our research on the view-based approach to automated reverse engineering of software architecture models, it is evident that our investigations will be extended in scope. This expansion aims to incorporate more external case studies, enhancing the technological and architectural diversity of the systems under analysis. Such an expansion is essential for evaluating the applicability and accuracy of our reverse engineering techniques across a range of technologies and programming paradigms.

The investigation will encompass other programming paradigms beyond the currently dominant object-oriented and structured programming frameworks. The objective is to evaluate the applicability of the reverse engineering approach by integrating systems

developed under functional, reactive, or even agent-based paradigms. This will improve the refinement of our Retriever approach to accommodate various architectural styles and development practices. This integration will facilitate comprehension of how diverse programming constructs and design patterns impact the extraction and interpretation of software architecture from code.

The objective is to develop new extraction rules and model discovery techniques for different programming languages. This development is essential to address the peculiarities of other languages and frameworks, which may require the application of tailored approaches for architecture reverse engineering. The objective is twofold: first, to improve the precision of our existing approach, and second, to devise novel techniques that can consistently decode software architectures.

We aim to improve the applicability of attack propagation analysis and vulnerability annotation, representing aspects of software security. Integrating these techniques into our reverse engineering framework could facilitate comprehension of the security architecture of the analyzed systems. Validating these aspects will require refining the approaches to identify security-related architectural components and their interactions accurately.

The planned extensions and improvements are intended not only to validate and refine our Retriever approach's accuracy but also to support its relevance and applicability to modern software development challenges. By extending the scope of our research to encompass these elements, our objective is to provide a set of approaches for reverse engineering software architectures. This approach can be applied to diverse technologies, addressing emerging security concerns in software systems.

8.2.3. Construct Validity

It is essential to support construct validity to guarantee that the evaluation employed in research accurately reflects the concepts it is designed to evaluate. This aspect of construct validity is critical in case study-based evaluations, where supporting congruence between theoretical constructs and practical implementations is necessary for the conclusions' soundness. This subsection examines construct validity, focusing on three key aspects: the relationship between metrics and objectives, the specific metrics used, and the evaluation process and framework. This analysis aims to confirm that the construct validity of the case study-based evaluation is concentrated and reflective of the empirical intentions. Considering the factors mentioned above, it can be concluded that the evaluation's findings genuinely reflect the underlying theoretical propositions.

8.2.3.1. Relationship Between Metrics and Goals

In examining construct validity within our experimental framework, we emphasize the relationship between the metrics employed and the evaluation goals. This is done to support the reliability and reproducibility of the findings. This is important to support the

accuracy and reliability of the evaluation's conclusions. These metrics must align with the objectives of our view-based approach to automated reverse engineering, as this is important to interpret the accuracy of our architectural search and analysis endeavors. Any discrepancies between the intended evaluation goals and the metrics employed can result in misinterpretations of the suitability and applicability of the evaluation process.

The GQM approach represents an element of our validation, offering a structured process to support the idea that each metric is directly associated with a specific evaluation question and an overarching goal. This framework provides an approach to the design of our experimental evaluation, guiding the selection of appropriate metrics that directly reflect the research objectives. The application of GQM enables the formulation of precise and quantifiable objectives for each dimension of our Retriever approach. This supports the idea that the metrics utilized suit the specific dimensions under evaluation.

Applying the GQM approach allows an evaluation of the software architecture models derived from technology-induced views. To illustrate, metrics such as precision and recall can be employed if the evaluation's objective is to ascertain the accuracy with which particular architectural roles are identified within a software system. These metrics directly quantify the degree of correspondence between the identified roles and those defined in the actual architecture of the system, thereby providing empirical data for evaluating this specific aspect of the reverse engineering process.

This alignment of objectives and metrics is essential to validating the evaluation's findings and guaranteeing that conclusions are based on an empirical foundation. By establishing this link, the GQM approach reduces the risk of misinterpretation and increases the construct validity of our research. The GQM approach provides a framework for evaluating our reverse engineering approach to capturing and analyzing software architecture from multiple views.

8.2.3.2. Specific Metrics Employed

The primary metric used is precision and recall, designed to evaluate the accuracy of identifying and reconstructing architectural elements from technology-induced views. In our investigation of construct validity within the experimental evaluation of our view-based approach to automated reverse engineering, we have devoted attention to the specific metrics employed to evaluate its applicability and accuracy. These metrics have been selected to align with the objectives of our evaluation, thereby supporting the idea that each metric offers meaningful insight into the capabilities and limitations of the reverse engineering process. This metric evaluates the accuracy of identifying and reconstructing architectural elements, including components, interfaces, and other features. It is a widely used measure of the accuracy of the reverse engineering process.

To further refine our understanding of the accuracy of our Retriever approach, we employ additional metrics, including precision, recall, and F_1 score. These metrics are essential for conducting a complete analysis of the performance of reverse engineering techniques. Precision is a metric that evaluates the proportion of correctly identified architectural

elements out of all identified elements, thereby providing insight into the accuracy of the identification process. In contrast, the recall metric evaluates the proportion of actual architectural elements that are correctly identified, thereby providing insight into the completeness of the reverse engineering process. The F_1 score, defined as a harmonic mean of precision and recall, provides a single metric that balances both precision and recall, thereby offering a view of the overall accuracy of the process.

These metrics, commonly employed in architectural analysis and reverse engineering studies, were selected for their capacity to provide a framework for evaluating our reverse engineering approach. These metrics facilitate complete process evaluations, from feasibility to more accurate evaluations. The application of these metrics supports that our evaluation can provide quantification of the performance of our Retriever approach, facilitate comparisons, and identify potential areas for improvement in future iterations of the research. The structured approach to metric selection contributes to the construct validity of the evaluation, as it supports that the metrics used are well suited to measuring the specific constructs under investigation.

8.2.3.3. Evaluation Process and Framework

In examining construct validity, particular attention is paid to the evaluation process and framework employed in our experimental evaluation of automated reverse engineering of software architecture models. At the core of this process lies the implementation of the GQM approach, which is applied to establish a structured and quantifiable evaluation framework. This process guarantees that each component of the evaluation process is directly aligned with the overarching research objectives, thus establishing a transparent link between the execution and our Retriever approach's effectiveness.

Subsequently, metrics are defined for each research question, providing quantifiable measures to evaluate the reverse engineering process in achieving the identified goals. These metrics include, but are not limited to, precision, recall, and the F_1 score previously mentioned. The evaluation framework compares the anticipated architectural design, typically delineated in the original system documentation or architectural diagrams, and the model extracted through reverse engineering. This comparison is critical, as it evaluates how the extracted architecture aligns with the originally intended design and functionality.

The integration of the GQM approach ensures that the evaluation process is conducted within a sound framework, thereby enhancing the transparency and reproducibility of the results. This approach supports the idea that each phase of the evaluation is traceable and justified in terms of its contribution to the research objectives, thereby reinforcing the construct validity of the evaluation. This structured evaluation framework allows for an analysis of the results, thus enabling researchers to identify areas where the reverse engineering process is effective and where further refinement may be necessary. The evaluation process supports the conclusions' validity, thus providing a foundation for future research and applications in reverse engineering software architecture.

8.2.3.4. Usefulness for Quality Prediction

The experimental evaluation evaluates the applicability and accuracy of our approach, emphasizing its usefulness in improving both performance and security prediction models. We evaluated how our approach reduces manual effort and addresses common challenges in reverse engineering software architecture quality prediction models. Overall, this evaluation focuses on system performance prediction and attack propagation analysis.

Our approach automates the extraction of component behavior within the PCM for performance prediction, facilitating accurate system understanding and architectural decision-making. However, manual refinements are required to parameterize resource requirements, configure the environment, and detail workload specifications. These refinements ensure that performance models accurately reflect real-world conditions.

In the security domain, our approach automates the generation of system and deployment models, including security states and potential attack vectors. Attack propagation analysis requires manual specification of access control rights, refinement of vulnerability properties, and construction of attacker models. Integrating existing XACML policy specifications can reduce some manual effort by leveraging predefined access control configurations. Vulnerability property refinement involves adapting CVSS values to specific system contexts, while attacker model development requires detailed definitions of attacker skills, knowledge, and behaviors to simulate realistic attack scenarios.

Experimental results support the usefulness of our approach. Automated generation of repository, system, and deployment views significantly accelerates the reverse engineering process, allowing domain experts to focus on essential manual refinements. Our approach effectively combines automation with necessary manual input, thereby improving the generation and accuracy of performance and security prediction models within the PCM approach.

8.2.4. Reliability

The reliability of scientific research is contingent upon the consistency and reproducibility of results across trials and conditions. This guarantees that the findings are not merely the result of specific experimental configurations or transient circumstances. Instead, they are dependable representations of reality that other researchers can consistently replicate. To examine the reliability of our research processes, we have divided the evaluation into four subsections. The subsections mentioned above are as follows: reproducibility of results, environmental consistency, availability of resources, and subjectivity in evaluation. A complete evaluation of these subsections will facilitate an understanding of the various factors that contribute to the reliability of our case study-based evaluations. This, in turn, permits us to improve the replicability of our findings.

8.2.4.1. Reproducibility of Results

The reliability component of our experimental evaluation emphasizes the reproducibility of the results. The capacity of other researchers to reproduce results is a tenet of the scientific evaluation process, serving as a cornerstone of research validity. The evaluation employs widely accepted metrics, including precision, recall, and F_1 score, at various stages of the evaluation process. These metrics were selected for their utility in scientific research, where they are valued for their ability to provide unambiguous performance evaluations.

The application of standardized metrics improves the reproducibility of the results. This approach allows other researchers to adhere to the same evaluation processes, utilize the same metrics, and anticipate comparable results under analogous conditions. Such reproducibility is critical for validating the presented results and subsequent studies that build upon this foundational research without ambiguity. This approach thus contributes to the reliability of the evaluation, supporting the idea that the results can be independently replicated, thereby confirming the research conclusions.

8.2.4.2. Environmental Consistency

In our applicability evaluation, we pay particular attention to the environment's consistency, which is essential for reproducing research results in reverse engineering software architecture models. Documentation of the software environment, including the employed approach versions and configurations, is critical. Variations in software environments can result in discrepancies in research outcomes. Therefore, it is necessary to regulate these variables to guarantee the integrity and reproducibility of the results.

To this end, our experimental evaluation records every aspect of the computing environment utilized during the experiments [Kir+24d; Kir+24c] to support the highest reproducibility standards. This encompasses the versions of software approaches used and the operating systems, underlying hardware specifications, and network settings, where applicable. Such documentation supports the idea that other researchers can recreate the exact conditions under which our evaluation was conducted, which is essential for verifying results through replication.

The documentation includes the configuration settings of the utilized approaches, such as the parameters for the reverse engineering algorithms and the precise configurations of any supplementary software that could potentially influence the outcomes. This level is intended to eliminate or reduce the potential for variability arising from differences in software environments when other researchers attempt to replicate the results.

This approach demonstrates a commitment to scientific standards. It improves the reliability of the evaluation by ensuring that environmental factors that could potentially bias the results are well-controlled and transparently reported. It establishes a foundation for future studies by providing a benchmark that can be used as a reference point for subsequent research in the field.

8.2.4.3. Availability of Resources

To improve the dependability of our investigation into the view-based approach to automated reverse engineering of software architecture models, we have identified resource accessibility as a consideration. In recognition of the role that data access and approach accessibility play in scientific research, our evaluation guarantees the public accessibility of all tools, source code, reference architectures, and software architecture models employed in our experiments. This commitment to transparency is intended to simplify the replication of our evaluation by providing other researchers with unrestricted access to all necessary materials.

By making these resources available [Kir+24d; Kir+24c], researchers can examine the evaluation processes and results obtained, facilitating a thorough understanding of the process. This open-access policy encompasses not only the data and software utilized, but also any supplementary materials that simplify the reproducibility of the research findings. For instance, documentation of the configuration settings, algorithm parameters, and precise versions of the software approaches employed is provided to support the idea that external researchers can accurately reproduce the environment of the original evaluation.

Providing reference architectures and software architecture models as part of the publicly available resources serves two purposes. Firstly, it permits researchers to corroborate the findings presented in the evaluation by directly contrasting their replicated results with the provided benchmarks. Secondly, it provides a foundation for further research and development in this field, offering a basic framework upon which future studies can be built. This approach simplifies the reproducibility of our current research and encourages ongoing innovation within the scientific community, thereby enhancing the work's overall impact.

8.2.4.4. Subjectivity in Evaluation

In our ongoing efforts to support the reliability of our research on the view-based approach to automated reverse engineering of software architecture models, we have focused heavily on minimizing subjectivity in the evaluation process. Although we use consistent metrics and provide descriptions of the experimental environment to promote objectivity, some aspects of our evaluation may still be influenced by subjective factors, particularly those involving the interpretation of the reference architectures from the case studies.

The subjectivity inherent in evaluating architectural elements can result from variations in the researcher's experience, affecting how data is interpreted. This variability is most pronounced when dealing with architectural constructs that may not have explicit representations or when the reference architectures are subject to varying interpretations due to their complexity or lack of documentation.

Our experimental evaluation employs several strategies to mitigate these potential biases. First, manual architecture models are the gold standard for validating automated methods.

The appropriate architectural style is selected, interface semantics are defined, and component diagrams are created using UML tools such as PlantUML. Architectural patterns such as n-tier and microservices are analyzed to improve the model. Verification involves working with stakeholders and aligning the model with existing documentation to ensure accuracy and reduce bias.

Despite these measures, eliminating subjectivity is challenging, especially in areas involving qualitative evaluations, such as software architecture. Therefore, our experimental evaluation openly discusses these limitations and suggests that the results be interpreted with an understanding of the potential for subjective influence, emphasizing the need to consider the context in which the evaluations are made. This acknowledgment emphasizes our commitment to transparency and improves the reliability of our research findings.

8.3. Answering Our Research Questions

In the following, we present our answers to our two research questions based on the results of the experimental evaluation. In the previous sections of this chapter (Sections 8.1 and 8.2), we discussed the implications of our findings and the validity of our research approach and results. This section explicitly summarizes the answers to the research questions based on the experimental evaluations' results.

RQ1: What is an applicable and accurate approach to reverse engineering software architecture for diverse artifacts across technologies?

In this thesis, we present our Retriever approach as an answer to this question. Our approach is applicable and accurate for reverse engineering software architectures for different artifacts across technologies. This approach introduces a view-based reverse engineering strategy that addresses the challenges posed by the heterogeneity of modern software systems, especially those using web services and microservices architectures. Our Retriever approach uses extraction rules to process diverse artifacts, including source code files, configuration files, and deployment descriptors in different programming languages and technologies. It uses model-to-model transformations, considering technology-specific relationships and concepts, to reconstruct multiple system views from these artifacts. These views include the software system's structural, behavioral, and deployment views. To ensure accuracy, our approach defines extraction rules tailored to specific technologies and allows precise mapping of software artifacts to architectural elements such as components, interfaces, dependencies, and deployment configurations. Experimental evaluations of our approach, conducted on real-world projects of varying size and complexity, have shown high accuracy, recall, and F_1 scores in identifying structural and behavioral properties. These results confirm the accuracy of our approach in producing valid architectural models that abstract away from the source code. Our Retriever approach is practical, scalable, and adaptable, processing large and complex systems in acceptable run times. Reducing the reliance on manual documentation processes, which are often time-consuming and error-prone, increases the efficiency of software development and maintenance.

RQ2: How can different views of a software system be integrated into a unified model that addresses multiple concerns?

To answer this question, this thesis presents a metamodel-independent framework that bridges the gap between model-providing and model-using processes. It combines model-driven processes to gather information while processing multiple views and resolving overlaps between them. Our model-driven composition and refinement step improves the integration of model-deriving processes such as reverse engineering and the extraction of views from different software artifacts. Different views of a software system can be integrated into a unified model that addresses different concerns using model transformations and composition techniques implemented in our Retriever approach. The process begins by extracting multiple views from heterogeneous artifacts through model-to-model transformations. Each view focuses on a particular concern of the software system, such as structural components, behavioral interactions, or deployment configurations. Each extracted view is refined using specific rules to support coherency and completeness within that view. This step resolves any incoherencies in each view. Our approach uses model transformation techniques to map related concepts across views. This includes aligning components, interfaces, and relationships that may appear different due to the heterogeneity of artifacts in different views. Model composition techniques integrate refined views into a unified architectural model based on an existing metamodel, such as the Palladio Component Model. This step resolves overlaps and incoherencies between the views to ensure that the unified model correctly represents the system architecture. The unified model addresses multiple interests by integrating the system's structural, behavioral, and deployment aspects. It enables stakeholders to analyze and predict system properties such as performance and security, considering the different interests of the stakeholders. This approach harmonizes the views extracted from heterogeneous artifacts into a coherent architectural model. This unified model supports understanding of the software system, supports quality prediction, and helps make informed decisions for system optimization and maintenance.

Part IV.

Epilogue

9. Related Work

The comprehension and administration of contemporary software systems represent a challenge in scientific and industrial contexts. A software system's architecture is essential, as it consolidates the system's structure and impacts its performance, scalability, maintainability, and security. To understand a software system's architecture, it is essential to explain its structure, its components, and the relationships between them. This complex process supports system resilience and is applicable in a heterogeneous technology environment.

This research addresses the issue of automatically generating architectural models from software artifacts, focusing on component-based systems. A feature of component-based systems is their modular structure, whereby components are designed for reuse across different system parts, frequently on diverse implementation platforms. The heterogeneous nature of software artifacts presents a challenge, which can vary in format, programming language, and semantic definition. The loosely coupled nature of these components introduces an additional layer of complexity, rendering it challenging to derive a coherent architectural model without understanding the interactions between these diverse parts within the system.

The heterogeneity of the artifacts, which encompass a multitude of technologies and platforms, necessitates an approach to accurately comprehending the interactions between the diverse components. It is essential to accurately capture the system's functionality and ensure that the architectural model precisely reflects the software's operational reality. Conventional architectural extraction techniques are sometimes incomplete in addressing the diversity and intricacy of contemporary software systems, resulting in incomplete or inaccurate models.

In response to these challenges, we propose a novel approach, our Retriever approach. This approach employs reusable concept descriptions to identify and extract diverse architectural views – structural, behavioral, and deployment – from software artifacts. The objective of our Retriever approach is to synthesize these diverse views into a unified architectural model that accurately represents the system's architecture. This is particularly advantageous for software developers and architects who require understanding or redesigning the underlying structure of complex systems.

Our approach can be classified into three principal categories: model-based engineering, reverse engineering of software architectures, and knowledge composition. Model-based engineering involves creating and manipulating architectural models as artifacts throughout software development. Reverse engineering is the process of extracting architectural

elements from existing systems to understand their structure and behavior. Consolidating diverse data into a unified knowledge base is essential to comprehending systems, particularly those comprising multiple technologies and platforms.

An examination of existing approaches as related work reveals that, while there are similarities in the objectives pursued, our approach combines elements from these three categories to address the specific challenges posed by heterogeneity and component independence. Our Retriever approach provides a framework for the construction of accurate software architecture models by integrating three key categories: model-based engineering, which emphasizes model handling; reverse engineering, which focuses on the extraction of hidden details; and knowledge composition, which adopts a view of diverse data.

> The subsequent sections are based on the following authored publications: [SK19; Kir21; Kir+23a; Kir+23b; Gst+24; Kir+24a; Kir+24b]

9.1. Classification of our Approach

This section provides an overview of the various approaches and methodologies associated with reverse engineering, architectural reconstruction, and erosion control in software systems. It also addresses the classification and implementation of multi-view specification environments. These approaches and methodologies manage software systems, support architectural integrity, and accommodate evolving requirements through analysis and foresight.

In the evolving field of software engineering, distinguishing our Retriever approach is essential for clarifying its attributes and positioning it within the research landscape. We have classified our approach using existing taxonomies, highlighting its characteristics compared to other approaches. These existing taxonomies provide a structured framework for comparing approaches based on goals, processes, inputs, techniques, and outputs. This classification allows for analyzing the specific problems addressed, the solutions offered, and the novel techniques employed. These include integrating different views of system artifacts and using explicit trace links to maintain coherence in view specifications.

This comparative classification is valuable for academic research and practical applications, providing stakeholders with insight into different approaches' relative advantages and limitations. By categorizing our Retriever approach, practitioners, and researchers can quickly identify and compare them in literature reviews, meta-analyses, and database searches, ensuring they are considered stakeholders as they explore appropriate approaches. In addition, taxonomies standardize the terminology and criteria used to describe approaches, facilitate effective communication among stakeholders across geographic and professional contexts, and ensure a comprehensive understanding of our approach and its potential applications.

In conclusion, categorizing our approach within existing taxonomies improves its visibility and comprehensibility while solidifying its scientific and practical relevance. This classification defines our place among software engineering approaches and promotes recognition and adoption in appropriate contexts.

9.1.1. Reverse Engineering Approach Taxonomy

The survey conducted by Tonella et al. encompasses a range of reverse engineering techniques, focusing on architectural recovery, behavioral recovery, design recovery, and clone detection. The efficacy of these approaches is evaluated through various empirical studies, including testimonials, case studies, and controlled experiments. Notably, the survey results indicate that only 26.5 % of the 260 papers included in the study employ case studies, suggesting that this empirical method is used with some infrequency [Ton+07].

In addition, the authors present a taxonomy of reverse engineering approaches that allows this approach to be distinguished from other related approaches. The following section presents a comparative analysis of the Retriever approach with other work that addresses the relevant research areas related to this approach and includes case studies. Sections 9.2.1, 9.2.2 and 9.2.3 account for the work mentioned above within their respective research areas [Ton+07].

Method or Tool Is it a method or a tool?

Dynamic or Static Does it involve execution, or is it based on the code?

Input What input does it require?

Output What output does it produce?

Interaction What types of interactions does it support?

User Guidance What guidance does it require from the user?

Task Applicability What tasks can it be applied to? Is it general or specific?

Scalability What system size can it be applied to?

Our Retriever approach is a static tool designed to be used with a fixed set of inputs and outputs. Our approach requires the input of a wide variety of data, including software artifacts in various programming languages or configuration files. It produces unified architectural models through UML diagrams and quality prediction models. In this way, our approach allows for multiple forms of interaction, including quality prediction and documentation. It applies to various tasks involving reverse engineering component-based software architectures from various heterogeneous artifacts. It is practically scalable and can be used in different sizes, including large and complex software systems. Our Retriever approach is distinguished by the automated extraction and integration of diverse views, including diverse artifacts such as source code or configuration files, into architecture models suitable for quality prediction. This requires the development of a

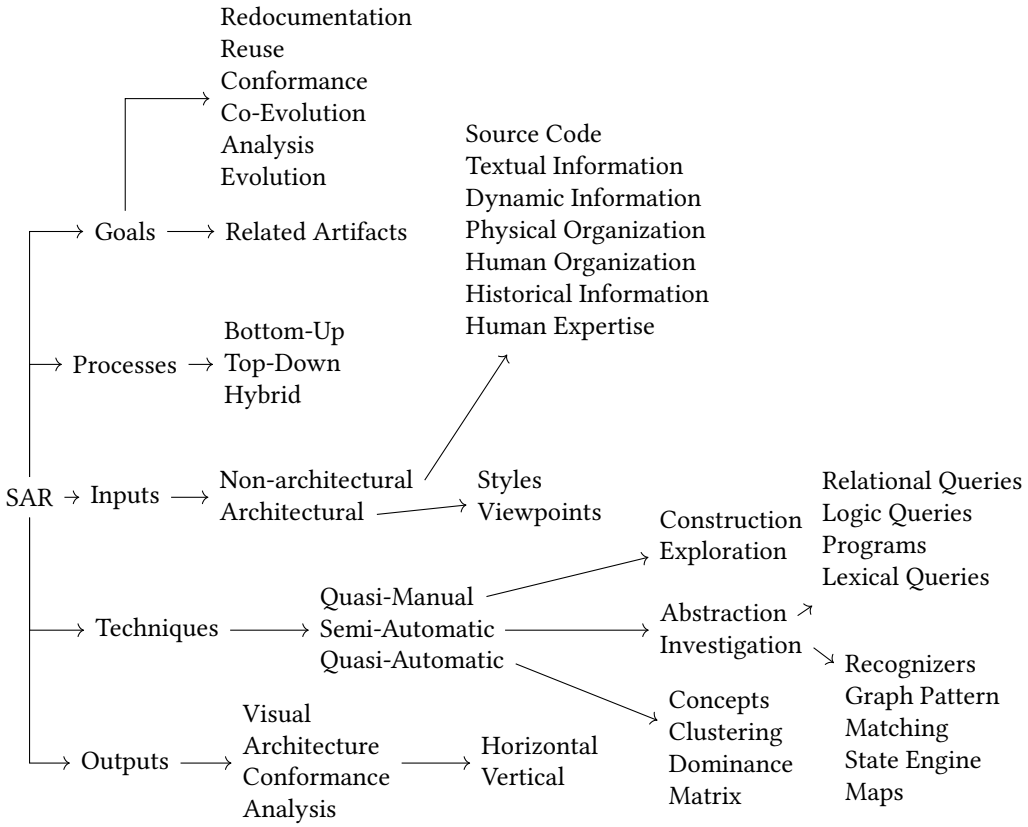


Figure 9.1.: Ducasse2009 Software Architecture Reconstruction: a Process-Oriented Taxonomy

knowledge representation model capable of accommodating the inherent heterogeneity of formats, languages, and semantics present in software artifacts.

9.1.2. Software Architecture Reconstruction Taxonomy

Ducasse et al.'s taxonomy provides a framework for classifying Software Architecture Reconstruction (SAR) methodologies along five axes. These categories are goals, processes, inputs, techniques, and outputs. This framework allows for comparing and differentiating different SAR methodologies. The taxonomy is illustrated in Figure 9.1, which shows the interrelationships among the processes, inputs, goals, techniques, and outputs involved in SAR. The visual representation provides a valuable visual aid for understanding the interrelationships among the various components of SAR [DP09].

Our Retriever approach follows the taxonomic framework developed by Ducasse et al. In particular, the objectives of our approach are oriented towards the documentation of existing material and extend to the domain of quality prediction analysis. This situates our approach within the Goals axis of the taxonomy, underscoring our emphasis on improving the understanding and prediction of prospective quality problems in software architectures. Regarding the Process axis, a bottom-up methodology characterizes our approach. The process starts with the software artifacts in their original form and proceeds incrementally

to construct an architectural representation. This process allows for an investigation, beginning with the components and progressing to creating an architectural structure. Our approach uses software artifacts as input, consisting of source code or configuration files in various programming languages. These inputs provide the data necessary for the initial stages of the architectural reconstruction process. Our approach can be classified as quasi-automated in terms of the techniques employed. The approach uses a combination of abstraction and exploration techniques facilitated by model-to-model transformations implemented by logical queries and pattern recognition. This allows for the extraction and refinement of architectural components. The outputs generated by our approach include visual representations of the architectural design. The outputs are designed to provide horizontal and vertical views of the architecture, thereby facilitating insight into the alignment between the intended and implemented architectures and identifying areas of non-conformance.

9.1.3. Architectural Erosion Control Classification

Silva et al.'s foundational study on software architecture erosion examines the divergence between planned and actual software architectures over time. The study methodically examines the strategies currently employed in industry and academia to counteract or mitigate such erosion. The research aims to formulate a classification framework for categorizing and evaluating the various architectural erosion control strategies. Figure 9.2 offers a visual representation of this classification framework of architectural erosion control approaches. The analysis identifies three approaches: minimize, prevent, and repair. Minimization strategies involve the identification of architectural deviations after implementation, thereby enabling the prevention of systemic problems. Prevention strategies are designed to avoid architectural erosion by strictly adhering to architectural plans developed during software development. In contrast, repair strategies correct identified deviations and realign the implemented architecture with the original design [SB12].

Our approach aligns with these classifications and presents a process-oriented architectural compliance strategy that incorporates elements of the three approaches above. The system supports preventive measures through continuous monitoring of architectural compliance coupled with quality prediction. This supports the proactive maintenance of architectural standards throughout the software development lifecycle. The trace-link model facilitates the identification of discrepancies between the intended and actual architectures by mapping the elements of one to the system artifacts of the other. In cases where erosion has been identified, the approach also serves a corrective function by providing mechanisms for recovering or discovering the original architecture. This can then be used to guide the restructuring of the software system to reflect its intended design. This integration facilitates the maintenance of architectural integrity and enhances system sustainability and adaptability in response to evolving requirements.

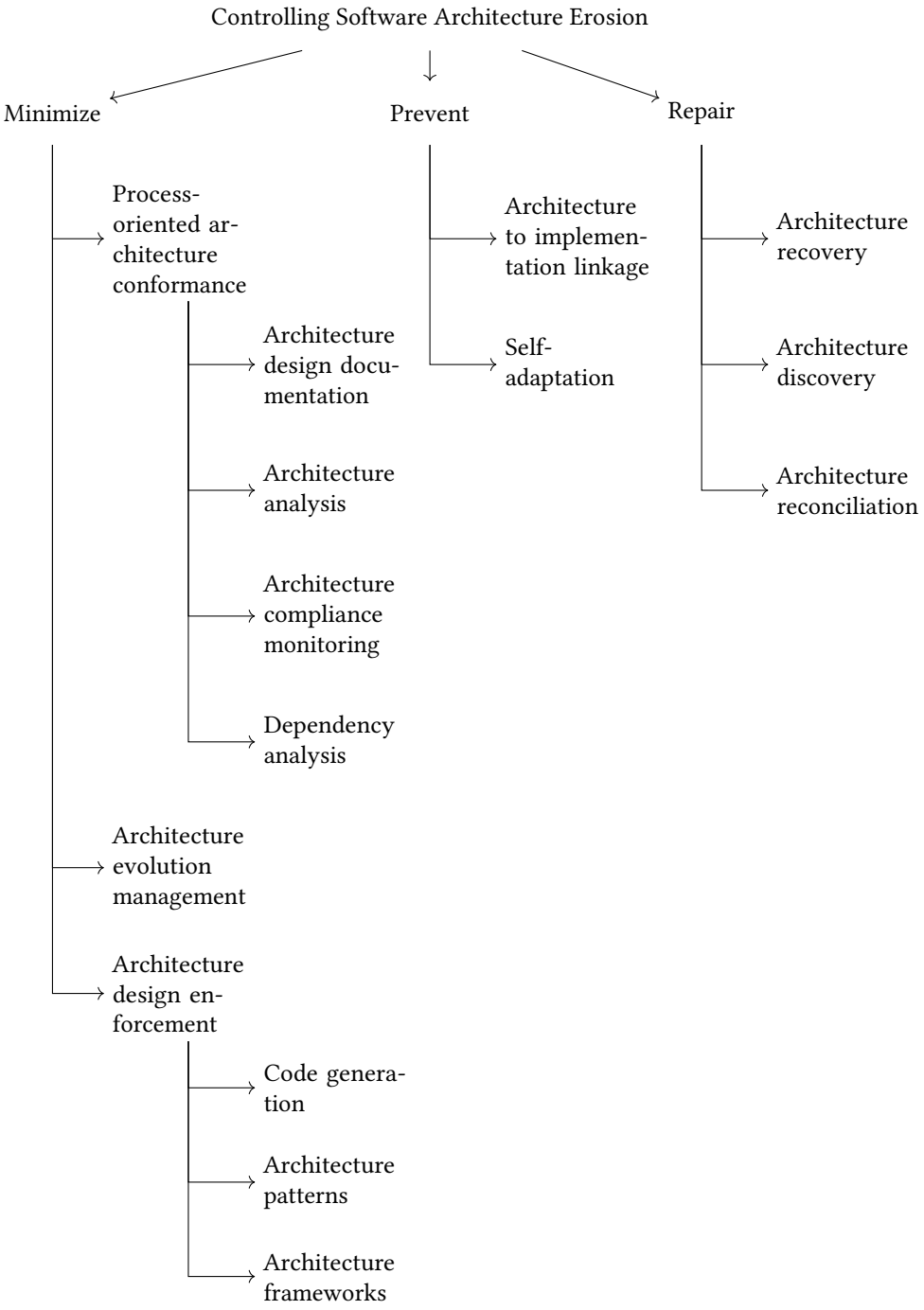


Figure 9.2.: Classification framework for existing architectural erosion control approaches presented by Silva et al. [SB12].

9.1.4. Multi-view Specification Environments Strategies

In their study, Atkinson et al. present several strategies for implementing multi-view environments in software engineering, [ATM15]. Our approach can be classified according to the five dichotomies. Our Retriever approach is consistent with this concept, which involves integrating different views by examining system artifacts and component-based architectures.

Rigorous versus Relaxed A rigorous approach requires the establishment of criteria for maintaining consistency across views and for delineating the content to be included in those views. In contrast, less rigorous approaches use multiple views without providing specifications for maintaining consistency or the content to be included in those views. Our Retriever approach is classified as relaxed because it does not provide specifications for maintaining consistency between views and models.

Synthetic versus Projective Views Projective views are derived from an underlying model that supports consistency. In contrast, synthetic views require manual maintenance of consistency between them. Our Retriever approach is both synthetic and projective because the artifacts must be kept consistent manually (synthetic). However, the architectural views are derived (projective).

Explicit versus Implicit Correspondences This dichotomy assesses whether the relationships between elements in different views are explicitly defined by inter-view correspondences or implicitly indicated by inter-view pointers. Our Retriever approach uses explicit correspondences by establishing trace links between views via a trace link model.

Extensional versus Intensional Correspondence In contrast to the extensional approach, which defines correspondences at the instance level and directly between views, the intensional approach defines correspondence rules at the type level. Our approach is extensional because it describes trace connections at the instance level and directly connects elements between views.

By employing these classifications, our Retriever approach provides a multi-view environment that takes advantage of explicit correspondences and projective views while maintaining a pragmatic structure that accommodates redundancy. It facilitates comparison and differentiation from related work, such as the coevolution approach. This Coevolution approach proposed by Langhammer is a multi-view approach that uses three different views: a source code view, a UML class diagram view, and a component-based architecture view. This approach is characterized by its emphasis on rigor, mixed explicit representation, intensionality, and pragmatism. It defines the necessary consistency between views, uses explicit correspondences and type-level correspondence rules, and incorporates redundancy through a virtual SUM [Lan+16; Lan19].

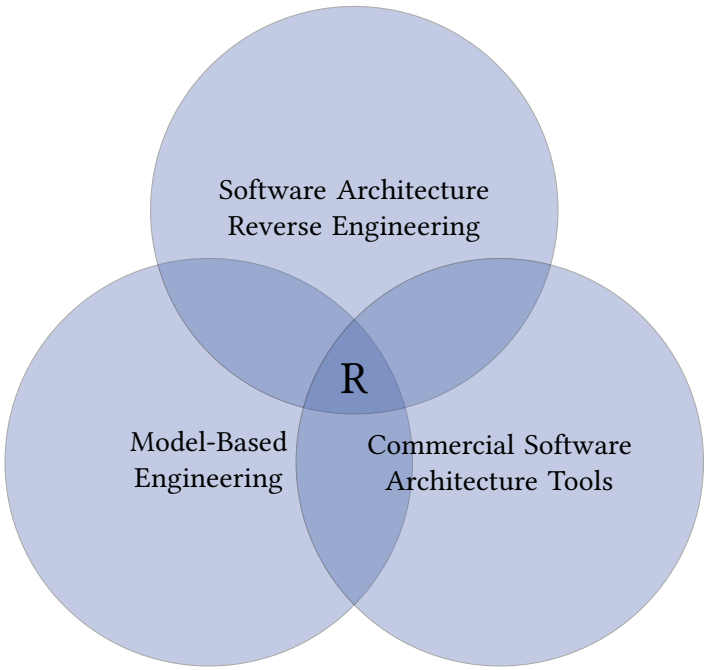


Figure 9.3.: This diagram illustrates the interconnections between our related work. The intersections identify potential points of convergence between research areas that may warrant interdisciplinary investigation or practical application. Our Retriever approach, abbreviated with the letter R, can be placed at the intersection of the three topics.

9.2. Relationships Between Related Works

The research areas listed for delimitation purposes demonstrate interconnections between the three topics and their associated sub-topics, making categorizing them exclusively within a single research domain challenging. Consequently, specific scientific endeavors may be situated within many research domains. The presentation of the overlapping research areas demonstrates the interdisciplinary nature of the listed topics, illustrating how different approaches can be combined to develop new solutions. To demonstrate the interconnectivity between the identified research areas, Figure 9.3 present instances where topics and processes converge or concepts from one area can be transferred to another. The following subsections analyze the relationships and interactions between the research areas.

The intersections in Figure 9.3 indicate potential connection points between the related works, which may interest interdisciplinary research or practical applications. Combining techniques from diverse fields represents a promising approach to resolving complex problems. The presented Retriever approach, based on reverse engineering and including the refinement and integration of the extracted views with elements and relations, can be considered an interface between the three topics in this context.

9.2.1. Model-Based Engineering

Our Retriever approach is based on the use of Model-Based Engineering (MBE), which represents a fundamental element of this approach. The objective is to develop an automated process for reverse engineering software architecture models from technology-induced architectural views. As an approach that emphasizes using models throughout the software development lifecycle, MBE provides techniques and principles that underpin our view-based approach. Our approach is distinguished by its employment of methodologies, specifically for the automatic generation and refinement of architectural models, to meet the contemporary needs of software engineering practice.

This section on related work addresses several aspects of MBE relevant to our thesis. Each subsection is dedicated to a dimension of MBE that aligns with our approach, explaining the continuity and innovations our work introduces to the existing literature. Each subsection presents related works that have shaped the current related works of MBE and provide a context for distinguishing our contributions. By examining this related work, we illustrate how our thesis aligns with other MBE practices and extends them to meet the challenges of reverse engineering software architecture in a technology-induced context.

9.2.1.1. Multi-View and View-Based Modeling

Kramer et al. present a view-centric approach that uses synchronized heterogeneous models through orthographic software modeling. This approach allows for the generation of adaptive views by combining multiple models that adhere to different metamodels and using generated model transformations to support consistency across views. The benefit is the ease with which new views can be created and maintained synchronized, eliminating the need for a monolithic metamodel. This is achieved by implementing a view-based reverse engineering approach to studying software architectures [KBL13; Kra19]. In contrast, our Retriever approach is based on separating concerns. Extraction is done based on specific technologies, while combination and refinement are done domain-specifically. Combination and refinement can also be considered independent of the technologies used. This allows the domain to include component-based systems, but the individual components can be considered independent of the technologies used. This allows the reuse of the combination and refinement steps in different component-based systems, where only the extractions need to be adapted to the technologies used in the system.

Atkinson et al. proposed a viewpoint language for projective modeling that allows views to be projected from an SUM, thereby leveraging deep modeling to span multiple classification levels. This approach eliminates the need to maintain pairwise consistency between views by ensuring that all views are consistent with the SUM. In contrast to our Retriever approach, which uses technology-induced views to reverse engineer software architectures, this framework prioritizes using projective views continuously updated under the SUM, reducing redundancy and maintaining consistency [AT17]. However, their framework faces difficulty managing updates and supporting code layout stability during reprojection.

Our approach addresses these issues by implementing a reverse-engineering approach and providing a composition and refinement approach for reverse-engineered architectural views.

Werner et al. introduce a generic language for generating query and view types, emphasizing a by-example approach to facilitate interaction with models by domain experts without requiring knowledge of query languages or model structures. The language is demonstrated using the role-based SUM and AutomationML [I62714], which illustrate a domain-agnostic language for defining queries and generating view types [WWA19]. Although this approach offers benefits for accessibility, it can present performance challenges when generating and updating views using a by-example mechanism. Our Retriever approach represents an alternative to the abovementioned language, providing a process for composing views based on predefined model structures.

Meier et al. present an operator-based approach for defining new views in software-intensive systems. In this approach, a series of small, reusable transformations (operators) are applied to an SUM, enabling the generation of customized views without requiring changes to the underlying model. The approach emphasizes iterative development, allowing stakeholders to focus on specific issues through customized views, facilitating updates to those views, and preventing information loss [MKW20]. However, the operator-based approach can present challenges in managing and configuring operators, potentially leading to performance bottlenecks. Our Retriever approach simplifies this process by using technology-induced views to reverse engineer software architectures, streamlining the process and supporting coherent updates with minimal manual intervention.

9.2.1.2. Model Composition and Refinement

Brunet et al. present a model merging framework that addresses the need to merge distributed models developed by different team members. The authors introduce the notion of treating the merge as an algebraic operator and describe the properties of a merge operator. The framework includes several related operations: match, diff, split, and slice. This approach emphasizes the explicit consideration of model relations as first-class objects. They illustrate the framework's application with examples and emphasize the importance of making explicit assumptions about model relations [Bru+06]. While this model merging framework is essential for managing distributed and heterogeneous models, it assumes that models are complementary and does not address scenarios with incoherencies. In contrast, our Retriever approach explicitly focuses on technology-induced views and reverse engineering of software architecture models. It provides an approach to dealing with incoherent and partial views. Our approach is based on model-driven foundations, thereby improving the resilience of our approach to the complexities encountered, where models may not always be perfectly aligned.

Sabetzadeh et al. present a framework for model merging that emphasizes treating relationships between models as first-class artifacts. This framework allows for specifying

and exploring various relationship types essential to the merging process. The framework is instantiated to support structural and behavioral state machine merging. The initial stages of the development process are concerned with structural merging. The framework uses graph-based techniques to resolve inconsistencies and abstract overlaps. Behavioral merging, applied in later stages of development, supports the preservation of semantic properties and addresses variability in model behavior through temporal logic [Sab+07]. Nevertheless, this approach provides a relationship-driven framework for model merging, which is advantageous for addressing syntactic and semantic relationships at different stages of development. However, it addresses state machines and relies on manual specification of relationships, which can be a cumbersome process for large models. Our Retriever approach focuses on reverse engineering software architecture models from technology-induced views, providing an automated approach to handling incoherencies and partial views. Our approach offers a solution for integrating architectural views and addressing merging and reverse engineering challenges.

Burger introduces the ModelJoin approach, which addresses the fragmentation of information across different metamodel instances in model-driven software development. ModelJoin allows developers to construct customized views using a human-readable textual DSL, thereby facilitating the integration of models without the need for manual model transformations. The approach includes an automated metamodel generator and higher-order transformations that enable the dynamic creation of views and support seamless access to relevant information. This reduces the complexity of model transformations and supports the creation of views, thereby improving traceability and consistency [Bur14; Bur+14]. The ModelJoin approach addresses the challenges of information fragmentation in model-driven software development by automating the creation of custom views. However, the approach addresses the integration of metamodels and does not address the reverse engineering of software architecture models from technology-induced views. Our Retriever approach builds on these strengths by providing a solution with reverse engineering capabilities. This enables the extraction and integration of software architecture models from different technology-induced views, addressing both the creation and reverse engineering aspects.

Bruneliere et al. introduce the concept of EMF views, which provide a mechanism for integrating heterogeneous models. This is achieved by defining views that aggregate elements from multiple metamodels without materializing the views. Such views redirect all model access and manipulation requests to the base models, supporting efficiency and synchronization. The approach uses a DSL based on SQL to define views, allowing the creation of intermodel views that behave like regular models. This mechanism improves the expressiveness and interoperability of the system while maintaining non-intrusiveness and synchronization with the original models [Bru+15]. The EMF Views approach addresses the challenge of heterogeneous model integration by providing a virtualized, non-materialized view mechanism. However, the approach does not address the issue of reverse engineering software architecture models from technology-induced views. Instead, it emphasizes view synchronization. In contrast, our Retriever approach develops these capabilities by integrating a reverse engineering process to extract and merge software architecture models derived from different technology-induced views. This supports an approach that

incorporates existing models and enables reconstructing architectural views from existing data.

Farias et al. proposed UML2Merge, an extension of UML designed to express merge relationships. This extension allows the definition of the semantics and order of merge relationships, thereby addressing UML's limitations in model merging. The UML2Merge extension introduces several novel constructs, including merge relationships, match strategies, and merge strategies, to facilitate accurate and automated merging of UML models. Empirical evaluation with developers indicates that UML2Merge is easy to use, produces accurate merge relationships, and is well received by users [Far+19]. The UML2Merge extension develops the ability of UML to handle merge relationships with precise semantics and order specifications, thereby improving the correctness and automation of model merging. However, its goal is to facilitate manual delineation and empirical validation of merge strategies within the context of UML rather than reverse engineering or managing heterogeneous technology-induced views. In contrast, our Retriever approach uses reverse engineering techniques to automatically extract and integrate software architecture models from technology-induced views, providing a solution encompassing both merging and reverse engineering. This addresses the challenges of software architecture reconstruction.

9.2.1.3. Model and Architecture Consistency

Langhammer et al. present Extract and EJBmoX that automatically generate models, including static architecture, behavior, and usage, from minimal system information, such as source code and test cases. This approach makes predicting performance possible without executing the system [Leo+15; Lan+16; Lan19]. These approaches are specific to enterprise Java technologies for mapping between source code and architectural models. They do not support software extraction with mixed technologies in a system. Our Retriever approach, on the other hand, refines and composes architectural models using multiple technologies. It also supports extracting other aspects, such as the automatic extraction of vulnerability models for analyzing a system's security properties.

König et al. present a framework and algorithm for checking the consistency of complexly interrelated models, thereby minimizing the use of expensive matching operations. The innovation is the technique of early localization, which reduces the data space subject to matching and merging. The approach is generalized to be applied to an arbitrary number of models. It incorporates formal approaches for specifying relationships through the use of partial mappings. The authors show that the localized-match-merge approach is more efficient than the traditional match-merge-localize approach, especially for large models with overlap [KD17]. This approach aims to check consistency in multimodel scenarios through early localization. In contrast, our Retriever approach takes a different approach and aims to extract architectural views by integrating multiple technologies. Our approach emphasizes extracting higher-level abstractions and supporting multiple stakeholder views, thereby addressing the challenge of heterogeneity.

Stevens address the complexity of maintaining consistency in model networks through bidirectional transformations. They extend the theory of bidirectional transformations to managing multiple models and describe the circumstances under which consistency can be maintained without interfering with transformations. They introduce a formal framework for consistency relations and propose algorithms for restoring consistency in large-scale model networks [Ste19]. Although this approach is theoretical and formal, it does not provide applicable solutions for integrating technologies and artifacts, as our Retriever approach does. Our goal is to provide an accurate solution to these challenges in reverse engineering. Our approach automatically reverse-engineers architectural models from heterogeneous sources, solving reverse-engineering challenges.

Klare et al. address the problems of fragmentation and inconsistency inherent in view-based modeling by implementing a model-driven development process, as proposed by the Vitruvius approach. They introduce the concept of virtual, unified models, allowing the integration of existing models without modification. The approach formalizes the process of maintaining consistency through the use of delta-based operations and consistency maintenance rules, thereby supporting coherence across multiple views within a system. The approach has been evaluated through embedded automotive software development case studies, demonstrating its benefits [Kla+21]. In contrast, our Retriever approach is designed to reverse engineer architectural models from different artifacts by integrating different views of other architectural components for various stakeholders. We prioritize combining and refining views while maintaining coherence through automated rules. Our approach is based on separating concerns since extraction is based on specific technologies, while composition and refinement are domain-specific. This allows the domain to include component-based systems, but the individual components can be considered independent of the technologies used.

9.2.1.4. Specialized Domain Application

Glinz et al. present the Adora approach to object-oriented modeling, which differs from methodologies, such as UML, by using abstract objects rather than classes as the building blocks of the model. The Adora approach is characterized by a hierarchical decomposition and integration of all aspects of the system into a coherent model. They provide an overview of Adora's concepts, the language used to express them, and a conceptual framework for visualizing the model hierarchy [GBJ02]. The Adora approach shares certain conceptual similarities with our Retriever approach, particularly its emphasis on hierarchical decomposition and integrating different aspects of the system into a unified model. However, while Adora emphasizes using abstract objects and object-oriented modeling to specify requirements and design logical-level architectures, our approach is oriented toward reverse engineering software architecture models derived from existing systems. The advantage of our approach over Adora is its ability to work with existing software architectures by using technology-induced views to derive architectural models, which is essential for understanding and documenting component-based systems. In contrast, Adora does not explicitly support reverse engineering and is oriented towards

forward engineering processes. Consequently, our approach can complement Adora by addressing the challenges of reverse engineering that Adora does not cover, thereby providing a toolkit for both forward and reverse engineering tasks in software architecture. This distinction highlights how our approach can overcome Adora's limitations regarding reverse engineering capabilities.

Vittorini et al. present the OsMoSys framework for multi-formalism modeling. This approach addresses the limitations of using a single formalism to analyze systems by integrating different modeling formalisms and solution techniques. OsMoSys offers several features, including metamodeling for defining and integrating different formalisms, support for both explicit and implicit multi-formalism, and model composition. They address the issues of formalism, interoperability, and model reuse through compositional modeling [Vit+04]. The OsMoSys framework and our Retriever approach aim to improve system modeling by integrating different views. The OsMoSys framework provides a platform for forward engineering through multi-formalism modeling, allowing other aspects of a system to be modeled using the most appropriate formalism. In contrast, our approach is toward reverse engineering, where architectural models are derived from existing software systems through technology-induced views. While OsMoSys emphasizes forward engineering in model construction, our approach shows particular competence in handling existing software architectures, facilitating their comprehension and documentation. The specific strength of our approach lies in its reverse engineering capabilities, which fill a gap not addressed by OsMoSys.

Mens et al. present IntensiVE, a suite of tools for documenting and verifying structural regularities in source code. This suite of tools facilitates the maintenance of evolvability in software systems by verifying the conformance of implementations to documented structural properties. The IntensiVE tool uses the concepts of intentional views and relations to facilitate grouping source code entities and the documentation of their structural dependencies. The tool suite provides feedback on discrepancies between the documented structure and the actual implementation, as illustrated by a case study of a Smalltalk application [MK06]. The IntensiVE tool suite and our Retriever approach aim to preserve and understand the software architecture through structured analysis. The IntensiVE tool suite is designed to facilitate documentation and verification of structural regularities in source code, thereby supporting consistency between design documentation and implementation. In contrast, our approach emphasizes reverse engineering existing software architecture models from technology-induced views. Although IntensiVE is a tool for preserving and verifying structural consistency during software evolution, it does not focus on extracting architecture models from existing systems. Our approach addresses the limitations of IntensiVE by enabling reverse engineering and documentation of software architectures from existing implementations.

Bork et al. present an approach for maintaining synchronization between code and models in model-driven development. It addresses the limitations of traditional reverse engineering approaches that rely on source code parsing by using the same templates for code generation and reverse engineering. This approach facilitates maintenance by automatically adapting to template changes, eliminating the need for different language parsers

[Bor+08]. The goal of the template-based reverse engineering approach and our Retriever approach is to facilitate the synchronization and understanding of software models and source code. The template-based approach facilitates the reverse engineering process by reusing code generation templates and supporting the idea that template changes are automatically reflected in the reverse engineering process. This approach addresses the maintenance challenges when templates or code generation strategies are modified. In contrast, using technology-induced views, our approach is designed to extract software architecture models from existing systems. This makes it an approach for reverse engineering component-based systems or systems where the original models are unavailable or incomplete. While the template-based approach is effective at maintaining synchronization in a controlled, model-driven environment, the strength of our approach lies in its ability to reconstruct architecture models from potentially undocumented software artifacts. By addressing the limitations of the template-based approach in dealing with undocumented systems, our approach complements this approach. It provides a solution for large-scale reverse engineering tasks.

Voelter et al. present mbeddr, an extensible language and integrated development environment for embedded software development based on C. mbeddr uses the JetBrains Meta Programming System (MPS) language workbench [Cam16] to modularize and extend C with domain-specific constructs, including state machines, components, decision tables, and requirements tracing. They illustrate mbeddr's ability to facilitate the construction of custom tools for small organizations and present illustrative examples of its use and benefits in concrete projects [Voe+13]. Both mbeddr and our Retriever approach are designed to improve software development and maintenance processes by providing new approaches. mbeddr is particularly adept at extending the C language with modular, domain-specific constructs, thereby facilitating the creation of custom development environments. This approach is particularly advantageous in embedded systems, where domain-specific requirements are ubiquitous. In contrast, our approach is designed for reverse engineering software architecture models derived from existing systems. This capability is essential for understanding and documenting the architectural structure of systems with incomplete or missing documentation. The advantage of our approach is its focus on reverse engineering from existing codebases, a domain that mbeddr does not address. Although mbeddr provides forward engineering capabilities, our approach offers complementary tools for reconstructing software architectures derived from existing implementations. This distinction highlights the contribution of our approach to the maintenance and evolution of software systems by addressing reverse engineering challenges not addressed by mbeddr.

Brecher et al. present a modeling framework tailored for machine-as-a-service applications in manufacturing. This framework employs a multi-level and modular methodology for formalizing product specifications, manufacturing processes, and resource capabilities. The framework aims to facilitate agile, customer-centric manufacturing by distinguishing product characteristics and resource capabilities, which separate models represent. It incorporates standardized Systems Modeling Language (SysML) [SysML] and UML notations, facilitates ontology-based extensions, and illustrates its implementation in the assembly of transmission gears [Bre+18]. The Multi-Level Modeling Framework for

Machine as a Service and our Retriever approach is designed to improve system modeling through structured approaches. The Multi-Level Framework is particularly adept at forward engineering, allowing the integration of various standards and ontologies for modeling product specifications and resource capabilities in automated manufacturing. In contrast, our approach is oriented towards reverse engineering, where architectural models are extracted from existing software systems using technology-induced views. The advantage of our approach is its ability to address component-based systems and undocumented architectures not explicitly discussed by the multi-level framework. Providing tools for reverse engineering serves to improve the forward engineering capabilities of the multi-level framework, providing a solution for both building and understanding systems. This synergy demonstrates how our approach can overcome the limitations of the multi-level framework in reverse engineering, making both approaches valuable for managing software and manufacturing systems throughout their lifecycle.

9.2.2. Software Architecture Reverse Engineering

Our Retriever approach is based on the fundamental concept of Software Architecture Reverse Engineering (SARE), which aims to provide a view-based approach for the automated reverse engineering of software architecture models derived from technology-induced architectural views. SARE is a methodology for exploring the foundational structures of software systems, mainly when explicit architectural documentation is unavailable or outdated. Our approach completes the traditional SARE approach by employing techniques to extract and combine architectural views into coherent software architecture models. This addresses the shortcomings of existing approaches, which frequently encounter difficulties in developing automated and scalable solutions.

This section presents an overview of the field of SARE, structured to facilitate understanding. It provides an examination of the techniques that inform our approach and distinguish our applications from other ones. Each subsection presents specific related work that has influenced the field of SARE. Our thesis incorporates and extends practices to address the challenges associated with automated reverse engineering of software architectures in a technology-induced context.

The following six subchapters examine and define the work considered in this paper in the context of reverse engineering software architectures. Section 9.2.2.1 discusses instrumentation and dynamic analysis techniques, Section 9.2.2.2 focuses on dependency-based approaches, Section 9.2.2.3 examines rule-based approaches, Section 9.2.2.4 discusses machine learning-based approaches, Section 9.2.2.5 presents view-based approaches, and Section 9.2.2.6 examines model-driven approaches.

9.2.2.1. Instrumentation and Dynamic Analysis

Tonella et al. present a novel approach to reverse engineering web applications. This approach uses a dynamic analysis technique that analyzes HTML code generated during

execution. They emphasize the inherent challenges associated with static analysis, due to the generation of dynamic content and the necessity of overseeing the utilization of multiple programming languages. Their approach entails pre-specifying input values for extended navigation coverage and generating a UML model augmented with statistical data from web server logs for subsequent analysis [TR02].

Brosig et al. present an automated approach for extracting architecture-level performance models for distributed component-based systems utilizing runtime monitoring data. The approach demonstrates a high degree of accuracy in performance prediction. However, this approach necessitates monitoring and data aggregation, which may introduce overhead, and relies on instrumentation for adequate control flow and resource demand analysis [BHK11].

Granchelli et al. present MicroART, a tool designed to reconstruct the architectural framework of microservice-based systems. The MicroART tool generates physical and logical architecture models by integrating static analysis from source repositories with dynamic analysis of runtime logs. The approach is distinctive in that it identifies and eliminates service discovery services, thereby disclosing the actual microservice dependencies and enhancing comprehension and examination of the architectural framework [Gra+17].

Heinrich et al. present an approach for architectural runtime modeling and visualization to support quality-aware software development in cloud applications. The approach integrates development models with runtime monitoring, enabling continuous model updates and facilitating collaboration between developers and operators. This approach promotes improved communication and cooperation between the development and operations teams. This is accomplished by maintaining semantic relationships between monitoring results and architectural models and combining descriptive and prescriptive models to facilitate improved communication and analysis [HZJ17].

Walter et al. present an extensible framework for extracting architectural performance models from monitoring log files. The framework is designed to be applicable across various architectural modeling languages, facilitating the reuse and integration of performance model extraction tools. This enables developers to implement particular model builder interfaces, thus reducing the construction of performance models for various languages, including the PCM. The framework integrates monitoring, log processing, and resource estimation tools, providing accurate performance predictions and reducing the complexity of implementing new extraction tools [Wal+17].

Eismann et al. propose an approach for modeling parametric dependencies at runtime in component-based software systems. The approach addresses three shortcomings of existing models: dependencies at the component instance level, the presence of multiple descriptions for a given variable, and the lack of explicit representation of correlations as parametric dependencies. Incorporating these attributes into a modeling language allows for the precise prediction of performance, which is important for capacity planning and proactive auto-scaling [Eis+18]. In contrast, our Retriever approach is oriented toward reverse engineering software architecture models from technology-induced views, thereby facilitating static analysis without the necessity of runtime data. Our approach

employs technology-induced views to generate precise models while reducing the burden of runtime overhead and the intricacy of instrumentation.

Grohmann et al. present a framework for continuous and self-adaptive resource demand estimation. The system employs a combination of estimation approaches that are selected and executed dynamically to adapt to environmental changes. The evaluation demonstrates the capacity to adapt to the online trace, reduce model error through continuous optimization and selection processes, and exhibit promising results in realistic cloud environments [Gro+21].

Cortellessa et al. present a model-driven approach for continuous performance engineering in microservice-based systems. The proposed approach employs design-runtime interactions to monitor microservices and derive refactoring actions to improve performance. System performance is analyzed through distributed tracing, which allows for the correlation of runtime data with the architectural model [Cor+22].

Unlike the seven previous works, our Retriever approach uses a view-based approach that prioritizes technology-induced views for reverse engineering software architecture models. Our static approach reduces the need for runtime data collection and log file monitoring, thereby limiting the scope of analysis to the observed aspects. As a result, our approach minimizes computational overhead and improves scalability for large systems. By leveraging multiple views, our Retriever approach facilitates accurate and minimally intrusive architectural modeling without requiring continuous runtime monitoring or instrumentation. It also supports integrating diverse technologies into the model-building process and enables the detection of issues beyond runtime observations, such as potential security vulnerabilities. Our approach provides a scalable, adaptable, and less invasive solution for architectural modeling that addresses the limitations of instrumentation and dynamic analysis techniques that rely on runtime interactions and data collection.

9.2.2.2. Dependency-Based Reverse Engineering

As proposed by Garcia et al., the prevailing process in automated architecture reconstruction is based on clustering approaches for the organization of software entities. In the authors' analysis [GIM13], which evaluates some software architecture reconstruction approaches, the empirical evidence supports the claim that clustering is the most prevalent form of decomposing software systems into manageable components. By focusing on this dominant technique, the authors provide valuable insights into its effectiveness and pave the way for refining existing approaches or developing new strategies. This could potentially improve the precision and utility of automated architectural reconstruction.

Clustering Approaches Krogmann et al. presents an approach that uses genetic programming to construct parameterized behavioral models for predicting the performance of component-based software systems. This approach integrates static and dynamic analysis with genetic programming to generate models validated by byte-code benchmarking [KKR10; Kro12].

Tzerpos et al. work on the ACDC algorithm is based on a comprehension-driven approach to software clustering. This approach prioritizes the creation of clusters that facilitate comprehension of the software system. The algorithm employs patterns commonly observed in manual decompositions and ascribes meaningful names to clusters while maintaining cluster sizes that enable effective management. This approach differs from other clustering techniques that prioritize metrics such as cohesion and coupling, resulting in an inability to aid comprehension. The ACDC algorithm has demonstrated stability and performance in experimental settings, generating meaningful clusters for large-scale industrial systems [TH00].

Andritsos et al. present a scalable hierarchical algorithm for clustering categorical data in their work on LIMBO, which employs the Information Bottleneck framework. LIMBO can cluster both tuples and attribute values, thereby offering a compromise between efficiency and clustering quality. The algorithm generates a summary model for processing large datasets, employing mutual information as a quality metric. Although LIMBO performs superiorly to other categorical clustering algorithms, its dependence on information theory and complexity may restrict its applicability to use cases [And+04].

Mitchell et al. present the Bunch tool, a search-based clustering approach for automatically modularizing software systems. This Bunch tool employs a graph partitioning approach to disaggregate a graph of source code entities and their relationships into subsystems. A fitness function is used to evaluate the quality of the graph partitions, while search algorithms are utilized to optimize the clustering process. The Bunch tool has been designed to prioritize flexibility, scalability, and integration with other tools. It offers the capacity to oversee ubiquitous modules and facilitate user-driven clustering modifications. Although Bunch is a practical approach, relying on heuristic search does not always result in data clustering. The applicability of this approach is dependent on the initial random partitioning [MM06].

Unlike the four related works discussed previously, our Retriever approach extends software architecture reverse engineering by automatically deriving coherent architectural models from technology-induced views using rule-based algorithms. This addresses structural and technological aspects, overcoming the manual and pattern-based limitations of approaches such as ACDC and heuristic methods such as Bunch's search. In addition, our approach prioritizes the creation of quality prediction models. Designed to handle the complexity of modern software systems where technology drives the architecture, our automated solution improves applicability and accuracy compared to pure clustering algorithms.

Dependency-Based Approaches Fradet et al. addresses the issue of consistency across different architectural views in their work on consistency checking in multi-view software architectures. Their approach comprises a formal definition of views through graphical diagrams with assortments and a straightforward algorithm for consistency checking. They put forth a language of constraints for the exact delineation of consistency requirements and a decision procedure for validating these constraints [FLP99]. However, the scope of

their work is limited by their focus on structural properties and simple attributes, which may be inaccurate for addressing architectural concerns. In contrast, our Retriever approach goes beyond these limitations by integrating technology-induced views of software architecture and employing a framework for reverse engineering. Our approach employs rule-based algorithms to automatically generate structurally cohesive and coherent software architecture models across different technology domains.

Ducasse et al. present a reengineering environment that supports various tools and techniques for analyzing and visualizing software systems. The Moose framework is constructed around a language-independent metamodel that permits extensibility, exploration, and scalability. It offers various services, including grouping, querying, and navigating software artifacts, enabling the integration of visualization tools, evolution, concepts, and dynamic analysis tools. The environment is designed to facilitate reengineering tasks through extensible infrastructure [DGN05]. However, it may encounter performance issues when processing datasets. In comparison, our Retriever approach employs technology-induced software architecture views to facilitate the automatic extraction and reverse engineering of architecture models. In contrast to Moose, which prioritizes providing an environment for reengineering tools, our approach emphasizes automated and coherent model extraction. This approach supports the idea that the derived architecture models are cohesive and coherent across different technology domains, thereby overcoming the scalability and automation limitations that can arise in Moose's manual and tool-dependent framework.

Haitzer et al. present a design science research approach that integrates software architecture and source code to prevent architectural erosion and drift. The approach uses evolutionary styles, architectural expertise, and architectural reconstruction to guide incremental evolution processes and support consistency between architectural design and code [HNZ17]. Both address the challenges of managing and evolving software architectures using model-driven techniques. However, they focus on different aspects of the problem. They address architectural drift and erosion, while we focus on the complexity of extracting models from diverse technologies. We use view-based modeling to create a unified software architecture model from heterogeneous software artifacts.

Müller et al. present an open-source stack for unifying data sources for software analysis and visualization. Their approach integrates various software artifacts into a unified graph database, employing tools such as jQAssistant and Neo4j for data visualization. This stack enables the creation, storage, and querying of heterogeneous data, thereby addressing challenges associated with software analysis, including schema modeling, data selection, and data handling [Mül+18]. However, this work concerns Java-based artifacts, which may encounter limitations in addressing projects involving multiple languages and software artifacts. Our approach offers an automated reverse engineering software architecture model solution. By leveraging technology-induced views, our approach supports the precision and uniformity of architectural models across technologies. This approach can handle a range of artifacts and automates the extraction process, reducing manual effort.

Jordan et al. introduce the concept of a digital architecture twin in their work on AutoArx. This concept models and maintains up-to-date system architectures through automated re-

covery, integration, and co-evolution of architecture information. The framework employs data collection agents and recovery services to gather and consolidate architectural data from diverse sources, thereby supporting the continuous updating of the architecture twin following the evolving nature of the system. This approach addresses the challenges of maintaining current and consistent architectural documentation and supporting constant compliance checks and design recommendations [JLS22; Jor+23]. In contrast, our Retriever approach emphasizes the automated reverse engineering of software architecture models derived from technology-induced views, supporting completeness across different technology domains. In contrast to the AutoArx approach, which is concerned with maintaining documentation and integrating architectural information, our approach offers an automated solution for extracting accurate architectural models. It addresses various reverse engineering requirements and circumvents the constraints associated with manual and semi-automated documentation maintenance inherent in the AutoArx framework.

Singh et al. presents an approach for automatically extracting architectural and behavioral models of message-oriented middleware-based microservice systems using static analysis. This approach extends the existing SoMoX framework by adding support for asynchronous communication [SWK22; SK24]. In contrast, our Retriever approach is based on combining and refining architectural models derived from a set of technology-induced views. This approach provides adaptability to evolving systems and other architectural views. This approach emphasizes accurate component and communication identification, although the limitations of static analysis constrain it.

9.2.2.3. Rule-Based Reverse Engineering

Huang et al. present a rule-based approach for reverse engineering source code into graphical models. This approach translates programming language primitives into graphical model components, facilitating cross-language and cross-model flexibility. The approach was implemented to generate four COBOL source code models, including control flow and functional structure models [HSM92]. In contrast to this approach, which is constrained by the intricacy and verification challenges inherent to rule-based systems, Retriever offers a view-based approach that integrates many technology-induced views to construct software architecture models. This approach allows maintainers to capture software structures, circumventing the complexity of predefined rule sets.

Moore employs a rules-based approach to examine the reverse engineering of user interfaces in legacy systems. The study demonstrates that language-independent rules can identify user interface components in legacy code, facilitating partial reverse engineering process automation [Moo96]. Nevertheless, the approach is constrained by aliasing and terminal control semantics, which require dynamic analysis for resolution. In contrast, Retriever integrates structural and behavioral analysis, circumventing these limitations and providing a software architecture model. This approach supports the idea that aspects of systems are accurately captured, addressing the limitations of Moore's approach, which was static.

Klint et al. present RASCAL, a domain-specific language designed to analyze and manipulate source code. RASCAL unifies the processes of analysis and transformation at diverse levels, providing a unified approach to tasks that have previously required multiple processes. It supports various features, including pattern matching, abstract syntax trees, and traversal approaches [KSV09]. However, RASCAL's performance for large-scale applications and reliance on the user's ability to define custom data structures and algorithms suggest potential limitations in scalability and usability. Retriever addresses these limitations by offering a view-based approach that streamlines the incorporation of views into software architecture models. This approach improves the maintainer's ability to handle systems, obviating the necessity for customization or managing the performance bottlenecks inherent in the RASCAL approach.

Ichii et al. present a rule-based, automated approach to extracting software models from source code. This approach employs the MOF for model transformation, enabling configurable model abstraction based on user-defined rules. The program-oriented modeling framework generates intermediate models and applies user-defined abstraction rules before generating the final target model. Based on this framework, the approach can extract models from C source code for use with a model checker. This example demonstrates the flexibility and efficacy of the approach in reducing the burden of fault analysis in industrial software systems [Ich+12]. Retriever addresses the limitations of this approach by integrating multiple views of software architecture models, thereby providing an automated approach to reverse engineering for more than just one language. In contrast to this approach, which requires establishing rules and applying domain expertise, Retriever's view-based approach reduces the need for manual effort in extracting and abstracting models.

Alnusair et al. proposed a rule-based approach for identifying program code design patterns that employ ontology formalism and semantic rules. Their approach uses the semantic web to identify latent design patterns in software libraries, offering flexibility and effectiveness without needing hard-coded heuristics. Nevertheless, the complexity of ontology modeling and the necessity for precise semantic definitions may restrict the usability and efficacy of this approach, particularly in the context of large code bases [AZY13]. Retriever employs a view-based approach integrating multiple software architecture views to search for structures such as boundaries or connections between components. This simplifies the discovery process without ontology definitions, reducing the reliance on semantic rules and improving reverse engineering efficiency, particularly in large software systems.

Cai et al. present a design rule-based approach that facilitates reconstructing software architectures. This approach identifies and exploits stable design decisions over time and modularizes systems into subsystems. The proposed ArchDRH algorithm combines design rule-based clustering with traditional techniques, such as ACDC and Bunch, to improve the precision and efficacy of architecture reconstruction [Cai+13]. However, the complexity and specificity of design rules can restrict their applicability to systems with less formalized architectural patterns. In contrast, Retriever goes beyond this by integrating multiple technology-induced views to create software architecture models. This approach does not

rely on predefined design rules, allowing for applicability to various systems, particularly those with less formalized architectural patterns.

Lehnert et al. present a rule-based impact analysis approach for heterogeneous software artifacts. This approach integrates the use of UML models, Java source code, and JUnit tests. Their approach employs a unifying metamodel and a centralized model repository for managing dependencies and changes impact across artifact types [LFR13]. Nevertheless, the intricacy and particularity of the rule-based system may impede its scalability and the precision of impact detection, particularly in the context of heterogeneous software systems. Retriever addresses these limitations using a view-based approach that integrates multiple technology-induced views. This approach allows for software architecture analysis, reducing reliance on rigid rule-based systems and enhancing applicability across software environments.

Garzón et al. present an incremental and rule-based approach to reverse engineering object-oriented code in Umple, which fuses UML modeling constructs with programming languages such as Java and C++. The Umplificator approach automates this process, ensuring the system remains compilable at each step while incorporating modeling information. This approach facilitates program understanding and reduces maintenance expenses by unifying the model and code into one entity [Gar+14]. Nevertheless, this approach's efficacy is contingent upon predefined transformation rules, which may require adjustments to accommodate code bases. Retriever addresses the limitations of Umplificator's rule-based system by integrating multiple views and employing static and dynamic analysis. This provides a reverse engineering approach capable of handling various software systems without constant rule tuning. It also addresses the scalability and adaptability issues the Umplificator is prone to encounter.

Al-Obeidallah et al. proposed the Multiple Levels Detection Approach to recover design patterns from Java source code. This approach employs a structural search model that progressively constructs the structure of each design pattern and matches pattern signatures through a rule-based system. The approach demonstrates reasonable recognition accuracy [APK18]. However, it encounters challenges in identifying partially implemented patterns and necessitates static analysis, which can restrict its applicability to dynamic aspects of software. Retriever addresses these limitations by integrating static and behavioral analysis to construct complete software architecture models. This approach ensures that the software's structural and behavioral aspects are captured, thereby overcoming this approach's reliance on structural analysis and its difficulties with incomplete pattern implementations.

Genfer et al. present an approach for identifying technical and business cycle dependencies in microservice APIs using static code analysis. Their approach uses modular, reusable source code detectors to reconstruct communication models and define architectural metrics for identifying cycles [GZ21]. In contrast, our Retriever approach combines and refines architectural models derived from multiple views. We intend that the resulting models will facilitate the exploration of new avenues of analysis beyond the detection of cycles at interfaces.

9.2.2.4. Machine Learning-Based Reverse Engineering

Garcia et al. proposed a machine learning-based technique to facilitate architectural discovery. This technique emphasizes concerns, defined as a system's roles, responsibilities, concepts, or purposes. The approach described by the authors improves the precision and automation of component and connector identification by integrating concerns with structural data. This differs from traditional techniques, which rely on structural information and require manual effort to recover connectors [Gar+11]. In contrast, our approach employs a view-based automation technique that recovers software architecture models using technology-induced views that address structural and behavioral aspects. This all-around approach reduces the necessity for manual involvement.

Al-Obeidallah et al. presents a novel approach to reverse engineering legacy systems that use large language models to extract UML and OCL representations from source code. The approach promises to improve program understanding and modernization by integrating large language models for automated specification extraction that overcomes the limitations of conventional static and dynamic techniques [APK18]. Unlike our Retriever approach, which focuses on generating technology-induced architecture models for quality prediction purposes, this approach uses machine learning to translate source code directly into higher-level abstract models, with the potential for improved accuracy and efficiency. However, the effectiveness of this approach can be limited by the quality and specificity of the training data used to develop the large language model.

Guamán et al. presents a semi-automated reference approach to facilitate the reconstruction of software architectures. The approach integrates various reverse engineering techniques to reduce technical debt and improve software sustainability. The system employs machine learning to suggest architectural improvements based on iterative and interactive training on system datasets [GPD18]. In contrast, our approach automates the recovery of software architecture models from technology-induced views, focusing on structural and behavioral aspects. This provides an automated solution that minimizes manual intervention and addresses architectural drift and erosion issues.

Yazdi et al. presents a novel approach to reverse engineering the architecture of distributed systems. This approach uses process mining and data science techniques to focus on dynamic behavioral analysis. It employs minimal instrumentation to capture interactions within microservices, thereby providing insights into system behavior and performance bottlenecks based on real-world data [YP21]. In contrast, our view-based approach to automated reverse engineering uses technology-induced views to facilitate the automated reconstruction of software architecture models. Our approach, therefore, aims to produce models that can also be used for quality prediction. Our approach also reduces the manual effort required for instrumentation and data collection and addresses the challenges of scalability and complexity in distributed environments.

Brahmaleen Kaur Sidhu et al. presents a machine learning-based approach to software model refactoring. This approach addresses design defects with granularity by targeting functional decomposition in UML models. Their approach employs a deep neural network

to identify and address these issues through refactoring operations, thereby enhancing the overall quality of software design more efficiently than traditional localized refactoring approaches [BS22]. In contrast, our approach is oriented toward automatically recovering software architecture models from technology-induced views. This approach integrates the structural and behavioral aspects of software systems. In contrast to this model, which relies on machine learning to detect and address specific design flaws, our approach emphasizes all-around and automated architecture recovery.

Dehghani et al. have proposed a framework that facilitates the transition from monolithic architectures to microservices through MDRE and reinforcement learning. Their approach involves extracting models from legacy systems and applying reinforcement learning to optimize the mapping of code elements to microservices. This framework aims to reduce the necessity for manual effort and improve the efficiency of the migration process [Deh+22]. Our approach diverges from these models by concentrating on the automated recovery of software architecture models using technology-induced views. This technique addresses both the structural and behavioral aspects of the problem, thereby reducing the necessity for manual mapping and refactoring. In contrast to this approach, our approach offers an automated solution for maintaining architectural integrity and addressing architectural drift and erosion.

Keim et al. presents a novel approach for recovering trace links between software architecture documentation and code. They employ intermediate component-based software architecture models to facilitate the alignment of semantics between documentation and code. The approach unites two processes for establishing connections between documentation and models and between models and code. The challenge addressed by this approach is the semantic disparity between the documentation and the code artifacts. This gap is bridged through the use of intermediate models [Kei+24]. In contrast, our approach is oriented toward automating the reverse engineering of software architecture models from technology-induced views. This approach integrates structural and behavioral aspects, providing a solution for preserving architectural integrity and addressing architectural drift and erosion issues.

Lano et al. presents a model-driven approach to legacy system reengineering designed to facilitate the reuse of business-critical functionality and the porting of legacy code to modern platforms. The approach automates the abstraction of software systems to UML specifications and applies refactoring and architectural transformations to improve system quality and maintainability. The objective is to reduce the accumulated technical debt and support the semantic preservation of the original code's functionality [Lan+24]. In contrast, our approach automates the recovery of software architecture models using technology-induced views that focus on structural and behavioral aspects instead of a code-based approach.

Rukmono et al. proposed a deductive process for recovering software architectures that use large language models to facilitate maintenance by initiating the process from known architectural properties. They differ from inductive approaches, using a top-down approach, reference architectures, and chain-of-thought prompting with large language models to classify code units. The approach demonstrated satisfactory accuracy in a

proof-of-concept on an Android application, indicating the potential of natural language to facilitate explicable and precise architecture discovery [ROC24]. However, our approach automates the recovery of software architecture models from technology-induced views, providing coverage of structural and behavioral aspects.

9.2.2.5. View-Based Reverse Engineering

Kazman et al. present Dali, a workbench for extracting and fusing architectural views of software systems. The authors highlight the importance of extracting and integrating various architectural views from multiple sources, including source code, dynamic execution traces, and build files. This is done to understand the architectural intricacies inherent to a system. The authors illustrate the process with examples and demonstrate how Dali facilitates view fusion, thereby enabling the capture of architectural information that is not readily available from a single source. The authors assert that architectural documentation is an indispensable component for the analysis, reuse, and preservation of the integrity of software systems over time [KC98]. The authors' work shares similarities with our view-based approach, highlighting the necessity for a range of views to guarantee precise architectural reconstruction. However, as implemented by Dali, their approach relies heavily on human interpretation and integration of views, which can be time-consuming and susceptible to error. In contrast, Retriever automates the process of reverse engineering software architecture models by leveraging technology-induced views, thereby reducing the potential for human error and effort. Retriever introduces an approach to capturing and integrating technology-induced views. This provides a scalable solution to the problem of architectural understanding and documentation.

Richner et al. presents an environment for generating high-level, customizable views of object-oriented systems that combine static and dynamic information. The authors identify the limitations of traditional reverse engineering techniques, which are constrained by their reliance on predefined views and a focus on static or dynamic analysis. Their approach employs a logical programming language to construct bespoke queries that facilitate the generation of bespoke views, integrating static structural data and dynamic execution traces. A case study demonstrates the effectiveness of this process. The case study illustrates how the approach can facilitate comprehension of object-oriented software architectures through iterative refinement of views and an emphasis on the salient features of the system [RD99]. The authors' approach shares similarities with Retriever in its objective of providing an understanding of the software architecture through multiple views. However, its reliance on user-defined queries and a logical programming language introduces complexity and necessitates manual effort to adapt and refine views iteratively. In contrast, Retriever automates the extraction and integration of technology-induced views, thereby reducing the necessity for manual intervention and minimizing errors. While the authors focus on integrating static and dynamic data, Retriever builds upon this process by incorporating technology-induced views to capture architectural details.

Riva et al. presents an architectural reconstruction technique that integrates static and dynamic analysis, thereby facilitating the reconstruction of software architectures. The

approach described by the authors comprises four iterative steps: the definition of architectural concepts, the collection of relevant data, the application of abstractions, and the presentation of the reconstructed architecture. The technique employs hierarchically typed directed graphs for static views and message sequence charts for dynamic views. Features include manipulating and synchronizing static and dynamic views interactively, incorporating horizontal and vertical abstraction mechanisms, and using case management for logically partitioning traces. The environment developed for this technique provides a means of visualizing and understanding a software system's structural and behavioral aspects [RR02]. The authors' approach shares common goals with Retriever, particularly regarding integrating static and dynamic views to understand the software architecture. However, a lot of manual effort is required to define concepts and apply abstractions, and it relies on other tools for visualization and synchronization of results. In contrast, Retriever automates a portion of the reverse engineering process by focusing on technology-induced views that reduce the manual intervention required to collect and integrate architectural information. Although the authors provide mechanisms for abstraction and synchronization, our automated approach to capturing and integrating technology-induced views offers a scalable solution for reverse engineering.

Deursen et al. present Symphony, a structured, view-driven process for reconstructing software architectures. The authors present a process addressing the selection of views, integrating reconstruction techniques, and the overarching architecture reconstruction process. Symphony highlights the importance of identifying the most appropriate architectural views to address specific inquiries. A combination of static and dynamic analysis is employed to achieve this objective. The process is composed of three principal phases. The initial stage of the process is to define both the source and target views. Subsequently, mapping rules are applied. In the final analysis, the reconstructed views are subjected to interpretation to resolve any architectural issues [Deu+04]. Despite Symphony and Retriever being designed to facilitate accurate software architecture reconstruction, there are differences between the two in their approach and implementation. The Symphony approach is based on a structured process, informed by case studies. Consequently, manual input is required at various process stages, including defining viewpoints, establishing mapping rules, and interpreting results. Although this approach can be time-consuming, mainly when dealing with large systems and multiple views, Retriever automates the reverse engineering process by focusing on technology-induced views, thereby reducing manual intervention and minimizing errors and effort.

Tran et al. present a view-based reverse engineering approach to improve model interoperability and reusability within process-driven SOAs. The authors present an extensible toolchain that employs architectural views to facilitate the integration of models at diverse levels of abstraction. This approach allows the creation of bespoke views tailored to the specific requirements of different stakeholders and enables the generation of executable code from these views. This employs a modular approach to process modeling, wherein concerns are divided into discrete views. The partial interpreter pattern extracts information from extant process descriptions, providing high-level views for business analysts and low-level, technology-specific views for technical experts. This approach allows for the maintenance and evolution of process models and adaptability to changing

business needs [TZD08]. Each approach discusses using MDE principles to transform and manipulate models, emphasizing an automated view-based methodology for reverse engineering. They emphasize separating concerns and using different architectural views to manage complexity and adapt models to stakeholder needs. However, they address the complexity of managing and integrating different representations of business processes in SOAs by focusing on separating business and technical concerns. In contrast, our Retriever approach addresses the heterogeneity of software artifacts in microservice architectures by emphasizing integrating and reconstructing software architectures from different views. In contrast, our approach focuses on microservices and Web services, addressing the heterogeneity of artifacts – source code, configuration files – and extracting component-based architectures from multiple sources. It includes trace links, extraction, and refinement rules to unify different views into a single architectural model.

9.2.2.6. Model-Driven Reverse Engineering

It is important to contextualize our Retriever approach within the existing scientific work, focusing on the literature reviews conducted by Mushtaq et al. and Raibulet et al. Mushtaq et al. conducted a review, selecting 56 scientific works from an initial pool of 3820 works. This revealed shortcomings in the existing research on multilingual code analysis. These efforts demonstrate processes analogous to our approach, with MoDisco being similar [MRS17]. Likewise, Raibulet et al.'s review of 15 MDRE approaches reflects the versatility of these processes across application domains. In particular, they emphasize the applicability of MoDisco, which is closely aligned with the scope of our work [RAZ17]. These foundational studies provide a framework for understanding the current state of research and highlight the novelty of our contributions in addressing identified gaps and extending the foundations laid by these earlier works.

Johnson et al. present an approach for enabling interoperation between legacy information systems. This approach entails the reverse engineering selected system components to create a model that facilitates interoperation. This approach uses existing interfaces and focuses on relevant subsystems, supporting logical data independence [JR01]. However, the approach presented here is designed for changes to existing code and is limited in its ability to generate a model of the system architecture. In contrast, our approach employs a view-based approach to generate software architecture models from technology-induced views. This approach overcomes its limitations by providing a complete and adaptable system representation, allowing reverse engineering to support system analysis.

Di Lucca et al. present WARE, an approach to reverse engineering web applications that employ UML diagrams to model various aspects at diverse levels of abstraction. The approach emphasizes comprehension and preservation of web applications by creating structural and behavioral models. While WARE exhibits strengths, it encounters challenges when confronted with the complexities inherent to large web applications and the interdependencies between their constituent artifacts [Di +02]. In contrast, our approach is not constrained to the domain of web applications and offers a strategy for reverse engineering software architecture models from technology-induced views. Our approach

addresses the limitations of WARE by supporting the reverse engineering of large-scale systems and providing quality prediction capabilities, thereby increasing the scope of reverse engineering efforts.

Cosentino et al. present a MDRE approach for reverse engineering business rules from Java applications to improve system maintainability and adaptability. This approach employs modeling techniques to represent business rules at a higher level of abstraction, thereby facilitating their comprehension and manipulation by relevant stakeholders. However, its specificity to Java applications and business rules represents a limitation regarding its applicability to other programming languages and formats [Cos+12]. In contrast, our approach accommodates a range of programming languages and system components, not merely business rules. This applicability allows for reverse engineering software architecture models, thereby overcoming the specificity limitation inherent in this approach.

Sánchez Ramón et al. proposed a model-driven engineering process for reverse engineering legacy GUIs, emphasizing uncovering implicit layouts and generating explicit, high-level GUI models. While their approach offers assistance for modernizing GUIs, it focuses on layout extraction and lacks application to non-GUI components and different technology stacks [SSG13]. In contrast, our approach extends beyond the domain of GUIs to facilitate reverse engineering of the entire software architecture model. This approach offers a solution applicable to a range of system components, including both GUI and non-GUI aspects, thereby addressing the approach's limitations.

Brunelière et al. present MoDisco, an extensible MDRE approach for creating model-based representations of software artifacts. Supporting Java and XML technologies, MoDisco facilitates the generation of ASTs and corresponding models from source code and other artifacts. One of its advantages is its infrastructure, which separates technology-specific, scenario-independent, and use-case-specific components. This separation allows for adaptability to legacy system technologies and reverse engineering scenarios. MoDisco provides a foundation for MDRE within specific technology contexts [Bru+14]. However, it lacks the multi-technology integration and higher abstraction mappings required for software architecture modeling and quality prediction. MoDisco supports using Java and XML technologies to generate model-based views of artifacts. However, our specialized approach, which specializes in reverse engineering software architecture models for quality prediction, is not entirely supported by a more general approach such as MoDisco. In contrast, the MoDisco approach is designed to extract models from Java code and Jakarta Server Pages (JSP) and does not provide mappings to higher levels of abstraction. Although MoDisco is intended to be extensible regarding the technologies it supports, it does not directly offer a means to reuse a technology's common concepts. Furthermore, combining multiple technologies to create a unified view is impossible.

Reis et al. present XIS-Reverse, a MDRE approach to extract high-level system representations from legacy relational database schemas. The approach employs user-defined heuristics and transformations to improve the extraction process, discern generalizations and aggregations, and extract default values to augment the models. It should be noted, however, that the XIS-Reverse approach is limited in scope to extracting models from relational databases and does not address other types of artifacts, such as code or non-relational

data sources [RS17]. In contrast, our approach is not constrained by the specific data sources it can handle and thus can accommodate a range of software artifacts, including code and database types. This flexibility allows for a reverse engineering process, thereby overcoming the limitations of this approach, which is limited to relational databases.

Sabir et al. present Src2MoF, a MDRE approach that converts Java source code into UML class and activity diagrams. This approach incorporates an intermediate model discoverer and a UML model generator that operate in conjunction to extract and transform Java applications' structural and behavioral aspects. While Src2MoF is a practical approach for reverse engineering Java applications, it is constrained in its applicability to other programming languages and UML diagram formats [Sab+19]. In contrast, our approach is not constrained by specific languages or diagram types, facilitating a new reverse engineering process. This applicability permits the generation of high-level software architecture models from various source artifacts, thereby overcoming the inherent limitations of this approach, which is specific to Java.

Yang et al. describe C2AADL_Reverse, a MDRE approach for transforming multitask C source code into models, for code verification in safety-critical software systems. This approach encompasses transforming structural, behavioral, and runtime information with formal verification procedures employed. However, it does not employ supplementary code analysis techniques to augment architectural comprehension, focusing on verification over architectural insights [Yan+21]. In contrast, our approach integrates view-based approaches to generate software architecture models from technology-induced views. By incorporating code analysis, this approach supports verification and improves architectural understanding.

9.2.3. Commercial Software Architecture Tools

In their empirical study, Ali et al. examine the current practices, challenges, and requirements for maintaining architectural consistency in software systems. The authors identify several barriers to implementing formal approaches to supporting architectural consistency. The study of architectural consistency shows that tools such as Structure 101 and Lattix facilitate the use of multiple programming languages and extensibility. However, they lack standard support for heterogeneous systems. In addition, the study identifies various shortcomings of existing tools, including the difficulty of quantifying the impact of architectural inconsistencies, the limited visibility of such issues to customers, and the effort required to map systems to architectures. The study reveals a need for consistency tools to address these issues and provide real-time feedback during development [Ali+17]. Although the authors identify the challenges and barriers to adopting formal consistency approaches, our Retriever approach addresses these issues by providing a quasi-automatic, static tool for reverse engineering component-based software architectures. Whenever the software system is changed, an architecture model could be extracted to track the actual changes to the system architecture over time and, if necessary, compare them with the intended architecture. The result is a reduction in manual effort and an

improvement in the visibility and traceability of architectural inconsistencies. Our approach addresses these challenges practically, including tool support and integration with real-time development processes.

IBM Rational Rhapsody [Int24] is a model-driven development tool that facilitates the design of complex embedded and real-time systems using UML and SysML models. It can generate code, perform simulations, and integrate with other IBM tools. Rhapsody is a tool for visualizing complex systems and automating code generation. It enables the generation of UML models from which source code can be derived and facilitates the reverse engineering of existing code to allow its integration. However, it is essential to note that not all architectural models are automatically generated during the reverse engineering process [Int24]. A difference between our Retriever approach and IBM Rational Rhapsody is that the latter only allows language elements such as packages and classes. It does not use high-level architectural descriptions in the form of components. In contrast to IBM Rational Rhapsody, which is aimed at model-driven development for embedded systems and focuses on UML and SysML modeling, our approach offers a view-based reverse engineering approach that reconstructs high-level architectural models from existing systems.

Lattix Architect [Lat23] is a dependency mapping, analysis, and architectural modeling tool designed to help software architects understand complex systems, improve code quality, and facilitate architectural communication. It is particularly adept at identifying architectural violations and enabling refactoring. However, its effectiveness diminishes when dealing with large code bases and depends on data quality. Lattix can create models created using UML and SysML [Lat23]. However, because Lattix is concerned with static dependencies, it cannot provide insight into runtime behavior or performance issues and is therefore limited to static dependency analysis. In contrast to Lattix Architect, concerned with static dependency analysis and architectural enforcement, our Retriever approach addresses static and behavioral aspects by leveraging multiple technology-induced views.

Structure101 [Hea24] is a specialized tool designed to facilitate managing and improving complex codebases through visual representation, analysis techniques, and restructuring strategies. The software facilitates the understanding of the architectural structure of a given system, enables the identification of problems within the code, and promotes collaboration among team members. Structure101 can abstract a system into a language-independent graph for analysis purposes [Hea24; SVL08]. However, because of its focus on structural code analysis and visualization, Structure101 may not provide insight into behavioral aspects, such as performance issues or potential security vulnerabilities. Unlike Structure101 our Retriever approach integrates multiple technology-induced views to create architectural models that address quality prediction.

A review of the existing literature on software architecture analysis has indicated that our Retriever approach has new capabilities, especially in the context of existing taxonomies and approaches. This review illustrates the classification and implementation of our quasi-automatic, static tool for reverse engineering component-based software architectures derived from many heterogeneous artifacts, including source code and configuration files in

different programming languages. By leveraging technology-induced views, our approach generates unified architectural models, represented graphically by UML diagrams, and quality prediction models, represented by the PCM.

The reverse engineering process of the Retriever approach begins with aggregating software artifacts, which are then used to construct an architectural representation. Our approach is practically scalable and well-suited for large and complex software systems. The user must initially input a minimal amount of data; however, input is required to specify the extraction rules. Our approach aims to facilitate the documentation and quality prediction of software architectures. This approach is aligned with a taxonomic framework that prioritizes improving the understanding and prediction of quality issues within software architectures.

Further analysis of related work indicates that the Retriever approach offers advantages over existing approaches. By automating the integration of diverse software artifacts into architectural models, our approach reduces the need for manual effort and increases the visibility of architectural inconsistencies. An advantage of this approach is its ability to automate the reverse engineering process from many technology-induced views. This capability allows the creation of models that seamlessly integrate software systems' structural and behavioral aspects. This adaptability to different environments indicates a reduced dependence on manual effort, which contrasts sharply with other approaches that predominantly focus on isolated elements such as quality prediction, view management, or model merging.

The synthesis of the review results indicates that our Retriever approach improves the analysis of software architectures. It manages the diversity of software artifacts and the inherent complexity of component-based systems. This management capability could improve modern software architectures' understanding, management, and evolution. The resulting benefits include optimized system performance, streamlined maintenance processes, strengthened security measures, and facilitated development practices in complex software environments.

10. Conclusions

The thesis conclusion provides a final reflection on the research, summarizing its contributions, impact, and potential for future work in reverse engineering software architectures using the Retriever approach. This final chapter provides an overview of the research's key findings, implications, limitations, and suggestions for further research and development. The conclusion aims to conclude the research findings and identify potential avenues for future investigation and innovation in software architecture reverse engineering.

10.1. Research Scope and Boundaries

In scientific research, it is essential to acknowledge and delineate the assumptions, delimitations, and limitations associated with our Retriever approach. Transparency and integrity are critical to the credibility and reproducibility of the research. Assumptions (Section 10.1.1) are taken-for-granted beliefs that underlie the research, such as the expectation of the completeness and reliability of external sources. Delimitations (Section 10.1.2) are self-imposed boundaries that define the scope of the research, such as a focus on component-based software systems. Limitations (Section 10.1.3) are potential weaknesses outside the researcher's control, such as reliance on particular technologies and frameworks, that may affect the research findings. Identifying and stating these issues distinguishes scientific research from superficial projects and ensures that the research's design, scope, and potential biases are understood.

10.1.1. Assumptions

Our Retriever approach assumes that the selected technologies, such as Java, Spring, and Docker, and the selected case studies represent modern component-based software systems. This assumption is essential for the generalizability of the results; if the technologies and case studies do not accurately reflect the technology landscape of modern software development, the applicability of our approach to other systems or emerging technologies may be limited. For example, the existing extraction rules may not capture the characteristics of different programming languages or frameworks. To address this issue, it would be beneficial to periodically update the set of supported technologies to include emerging frameworks and platforms. Selecting a diverse set of case studies from different domains and with other architectures, would improve the relevance and applicability of our approach to a broader range of systems.

Our approach assumes developers adhere to standard practices and specifications associated with their frameworks and technologies. This assumption is essential because deviations from standards may result in incomplete or inaccurate architectural models. The extraction rules utilized in our approach may fail to capture unconventional implementations or non-standard coding practices, compromising the integrity of the reverse-engineered models. Building applicability into our approach to handling variations in coding practices would be beneficial to improving efficiency. Developing mechanisms to detect deviations from standard patterns and adjusting the extraction rules accordingly could help accommodate a broader range of development styles, leading to accurate models.

Finally, our approach assumes that external information sources, such as vulnerability databases and software artifacts, provide reliable data. This is important for building accurate models and conducting assessments of the software architecture. However, if these external sources are incomplete, outdated, or inaccurate, the resulting models may be flawed, potentially missing vulnerabilities or architectural elements. To improve reliability, it would be prudent to utilize multiple sources of information, implement regular updates to ensure the data remains current, and validate the data for accuracy. Working with the software development and security communities to improve these external sources could also be beneficial, as it would contribute to the overall quality and trustworthiness of the information on which our approach depends.

10.1.2. Delimitations

Our research focuses on reverse engineering software architecture models specifically for Component-Based Software Systems (CBSS), including web services and microservices. This specialization allows for an analysis of the challenges associated with these systems, such as modularization, service interactions, and practical scalability concerns inherent in CBSS architectures. However, this focus may limit the applicability of the findings to other architectural styles, such as monolithic or service-oriented architectures, which have different structural and interaction paradigms. Future studies could expand the scope to include these other architectural styles to assess the generalizability of our Retriever approach across different system types, thereby improving its relevance to a broader range of software architectures.

Our approach utilizes specific technologies and frameworks, such as Java, the Spring Framework, JAX-RS, Docker, Maven, and Gradle. It relies on components and interfaces explicitly defined by annotations or conventions. This reliance facilitates the extraction and analysis process by leveraging these technologies' standardized structures and metadata. However, it limits our approach to systems that utilize these specific technologies and adhere to these conventions, potentially limiting its applicability to other programming languages, frameworks, or systems that do not follow similar practices. Extending our approach to be technology-agnostic or to support additional languages and frameworks would improve its adoption in different technology environments, allowing for its applicability in the industry.

The PCM is employed as the ADL in our approach to building software architecture models and performing analyses such as attack propagation. While PCM provides modeling capabilities tailored to component-based systems, reliance on this specific ADL can limit our approach to domains where PCM is appropriate. Practitioners employing other ADLs may find it challenging to adopt our approach without adaptation efforts. Future research could generalize our approach to support multiple ADLs or develop an ADL-independent framework, thereby increasing its applicability across different modeling environments and facilitating wider adoption among software architecture practitioners.

10.1.3. Limitations

Our Retriever approach relies on particular technologies, frameworks, and technology-specific extraction rules and conventions. These rules are tailored to specific technologies and may not apply to other systems, limiting our approach's applicability and accuracy. Adapting our approach to new technologies requires effort and expertise to develop new extraction rules, which requires understanding the latest technologies and underlying architectural concepts. Developing technology-agnostic extraction mechanisms or adaptive rules could reduce this dependency, improve applicability, and allow application across diverse software systems.

Another limitation is the potential for inaccuracy and incompleteness in the reverse-engineered models. The process may not perfectly map code artifacts to architectural components, leading to misalignment and incomplete models, especially for elements not explicitly represented in the code, such as implicit dependencies or design decisions. This affects the accuracy of the architectural models, potentially leading to ineffective analysis or incorrect choices based on these representations. Improving our approach with manual validation steps, integrating additional sources of information such as documentation or developer insight, or utilizing machine learning techniques to infer missing information could improve model completeness and accuracy.

Another limitation of this work is that the approach has only been evaluated on some open-source projects. This sample size limits the generalization of findings to a broader range of software systems, and may not reflect the complexities of industrial applications. As a result, the applicability and accuracy of our approach in real-world scenarios, particularly in industrial settings, remain uncertain. Including case studies, particularly from industrial environments, would provide an evaluation and help validate our approach in real-world scenarios.

Our approach uses static code analysis and structure-level mapping rules to reconstruct architectural models from source code, effectively capturing the static structure, including components, interfaces, and interrelationships. However, this method may miss dynamic components and runtime behaviors such as dynamic binding, reflection, or configuration changes, potentially resulting in incomplete models. Relying solely on static analysis can

also lead to incomplete architectural models and security assessments by missing runtime-specific aspects such as dynamic module loading or runtime interactions, increasing the risk of undetected vulnerabilities.

10.2. Further Approaches and Future Work

A potential avenue for future work is to broaden the technology scope of our Retriever approach by incorporating support for a broader range of programming languages, frameworks, and architectural styles. A technology-agnostic approach can be applied to diverse software systems, increasing its applicability and adoption across different technology environments. This expansion would accommodate systems built with diverse tools and practices, allowing our approach to handle the heterogeneity that characterizes modern software development. Support for additional technologies would enable our approach to be employed in a broader range of industries and applications.

Another promising direction is to encourage the development of community-driven rule libraries and to promote collaboration between practitioners and researchers. This approach can benefit from the community's collective expertise by promoting shared rule libraries, resulting in rules considering different technologies and practices. This collaborative effort would facilitate knowledge sharing and ensure our approach remains current with the latest technological advances. Community input can also help identify and address common challenges, improving the approach's applicability and accuracy. Such collaboration enriches the rule base and encourages a sense of ownership and commitment within the community.

Another valuable extension of our approach is the integration of dynamic analysis techniques and incorporating runtime monitoring and runtime data. Static analysis alone may not capture all behaviors and configurations, especially those manifesting only during execution. By combining dynamic analysis, our approach can provide a complete architectural model that reflects the actual runtime behavior of the system. This integration allows the identification of dynamic components and the detection of vulnerabilities that may not be apparent through static analysis. Incorporating runtime data improves the accuracy of reverse-engineered models. It enables our approach to address performance issues, resource utilization, and other runtime characteristics important to quality prediction and system optimization.

The use of machine learning and artificial intelligence methods offers a promising way to improve the reliance of extraction on project- and technology-specific rules. Machine learning and artificial intelligence can automatically learn code and artifact patterns, which has the potential to streamline the creation and maintenance of extraction rules. This could reduce the amount of manual input required to create and maintain rules. The advantage is that the extraction process remains subject to the extraction rules, ensuring control and understanding of the extraction.

Conducting empirical studies in industrial settings is essential for evaluating our approach in real-world scenarios. By applying our approach to industrial software systems, researchers can assess its practicality and identify challenges that may not be apparent in open-source projects. Such studies can validate our approach's applicability to large, complex systems and uncover performance and security issues with existing workflows. Engaging with industry practitioners provides valuable feedback and enriches our approach's credibility among practitioners. These empirical evaluations can inform further refinements and adaptations, ensuring that our approach meets the needs of practitioners and aligns with industry best practices.

Including architectural documentation, design patterns, configuration files, and other non-code artifacts in the reverse engineering process can improve the completeness and accuracy of reconstructed architectural models. These artifacts capture implicit architectural decisions and design intent that are not apparent in the source code alone. By incorporating such non-code artifacts, our approach can closely align the reconstructed model with the original design intent, improving its applicability for stakeholders. This integration allows for a system architecture view, facilitating better understanding, maintenance, and software evolution. Including non-code artifacts also helps capture configurations and settings that affect system behavior, contributing to accurate quality predictions.

10.3. Summary

The thesis addresses the inherent challenges of modern software systems, characterized by technological heterogeneity and complexity, especially in web and microservice architectures. Our contribution introduced a view-based reverse engineering approach that extracts multiple views – structural, behavioral, and deployment – from different software artifacts. By integrating these views through model-to-model transformations and model composition techniques, the thesis constructs a unified, coherent architectural model that can be used for system analysis, optimization, and quality prediction.

This thesis contributes to the field by developing a novel technique for transforming reverse-engineered models into accurate representations that approximate the component-based software architecture of a system. Our Retriever approach employs a knowledge representation model tailored to heterogeneous artifacts and technology-specific extraction rules, thereby enabling the mapping of artifacts to architectural elements. This model-driven strategy supports extraction and captures information across multiple views, resolving inconsistencies and overlaps to produce an architectural model. The model thus provides an understanding of the system structure, enabling developers to maintain and optimize complex systems.

The second contribution is developing a framework for integrating the extracted views into a unified model using model transformation and composition techniques. This integration addresses the issue of architectural diversity by harmonizing different concerns, including performance and security, across multiple views. The unified model enables the application

of model-based quality prediction tools for evaluating and predicting system attributes such as performance and security. The empirical evaluation of our Retriever approach demonstrates its applicability and accuracy across real-world software projects, confirming its applicability and accuracy as a reverse engineering tool for software architecture.

The beneficiaries of this thesis are software architects, engineers, and researchers involved in developing complex systems, particularly those built using contemporary paradigms such as microservices. Our Retriever approach presents an automated approach for reconstructing and maintaining architectural models, representing an advance over current practices. By reducing the reliance on manual documentation and extending the scope of architectural analysis to include quality prediction, this thesis contributes to developing an approach that aligns system documentation with evolving codebases. Automated generation of architectural models supports consistent documentation of system properties, thereby reducing the risk of architectural drift.

The thesis impacts industry practices by improving the integration of architectural modeling into CI/CD pipelines, supporting continuous architectural validation, and preventing misalignment between system implementation and architectural design. This integration benefits software development teams, including improved collaboration, code quality, and early detection of architectural issues. In addition, the open-source implementation of our Retriever approach allows it to be adopted and adapted in various development environments, providing applicability for different technology stacks and project requirements.

Further research in this area could be pursued in some ways. In the short term, the technology-specific extraction rules could be refined, and the framework could be compatible with additional programming languages and platforms, thereby increasing its accuracy. As the evaluation showed, there were some limitations in mapping components that were not directly related to the system functions. Creating project-specific guidelines could address this issue and improve the accuracy of the approach in different scenarios.

In the longer term, efforts could be made to integrate machine learning techniques to predict potential architectural configurations and recommend improvements based on historical data patterns and system performance metrics. This would facilitate the development of predictive modeling capabilities, enabling systems to adapt to evolving requirements automatically. Extending the approach to behavioral modeling and security analysis could facilitate the creation of new application areas, such as automated threat modeling and vulnerability assessment. Incorporating security considerations into architectural models would provide stakeholders valuable insights for formulating mitigation strategies and improving overall system resilience.

Further research could explore extending the view-based approach to emerging software paradigms, such as serverless computing and edge computing architectures, which present new challenges in decentralization and distribution. Addressing these areas is imperative to support our Retriever approach, which remains relevant as software development evolves.

In conclusion, the research conducted in this thesis represents a contribution to the software architecture reverse engineering field, addressing the increasing complexity and

technological heterogeneity of modern software systems. The view-based Retriever approach provides an applicable and accurate solution for reconstructing architectural models and facilitating software system maintenance, optimization, and analysis. The thesis represents an advance in the state of the art by automating the documentation of architectural models and integrating them with modern development practices. It provides a foundation for future automated architectural analysis and quality prediction research. The approach's open-source nature and empirical validation position it as a valuable tool for academic and industrial applications, supporting its continued development and impact.

Bibliography

The titles of most entries are hyperlinks that resolve the Digital Object Identifiers (DOIs) or point to other online sources for the entries.

[AC18] Andrade, H. and Crnkovic, I. "A Review on Software Architectures for Heterogeneous Platforms". In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. Vol. 8. IEEE, 2018, pp. 209–218.

[Agh+19] Aghajani, E., Nagy, C., Vega-Márquez, O. L., Linares-Vásquez, M., Moreno, L., Bavota, G., and Lanza, M. "Software Documentation Issues Unveiled". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1199–1210.

[Agh+20] Aghajani, E., Nagy, C., Linares-Vásquez, M., Moreno, L., Bavota, G., Lanza, M., and Shepherd, D. C. "Software documentation: the practitioners' perspective. ACM/IEEE International Conference on Software Engineering". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Association for Computing Machinery, 2020, pp. 590–601.

[Ali+17] Ali, N., Baker, S., O'Crowley, R., Herold, S., and Buckley, J. "Architecture consistency: State of the practice, challenges and requirements". In: *Empirical Software Engineering* 23.1 (2017), pp. 224–258.

[All24] Allewar, A. *anilallewar/microservices-basics-spring-boot*. GitHub. original-date: 2017-04-20T10:48:46Z. 2024.

[Alo+04] Alonso, G., Casati, F., Kuno, H., and Machiraju, V. *Web Services: Concepts, Architectures and Applications. Concepts, architectures and applications*. Springer Berlin Heidelberg, 2004. 354 pp. ISBN: 9783662108765.

[And+04] Andritsos, P., Tsaparas, P., Miller, R. J., and Sevcik, K. C. "LIMBO: Scalable Clustering of Categorical Data. 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004". In: *Advances in Database Technology - EDBT 2004*. Springer Berlin Heidelberg, 2004, pp. 123–146.

[Apa24a] Apache Software Foundation. *Apache Commons CSV*. Version 1.11.0. Apache Software Foundation, 2024. URL: `https://commons.apache.org/proper/commons-csv`.

[Apa24b] Apache Software Foundation. *Apache Maven*. Version 3.9.7. Apache Software Foundation, 2024. URL: `https://maven.apache.org/`.

[APK18] Al-Obeidallah, M. G., Petridis, M., and Kapetanakis, S. "A Structural Rule-Based Approach for Design Patterns Recovery". In: *Software Engineering Research, Management and Applications*. Springer International Publishing, 2018, pp. 107–124.

[ASB10] Atkinson, C., Stoll, D., and Bostan, P. "Orthographic Software Modeling: A Practical Approach to View-Based Development. International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE". In: *Evaluation of Novel Approaches to Software Engineering*. Springer Berlin Heidelberg, 2010, pp. 206–219.

[Aso+24] Asomov, A. et al. *SnakeYAML*. Version 1.30. SnakeYAML Project, 2024. URL: `https://bitbucket.org/snakeyaml/snakeyaml`.

[AT17] Atkinson, C. and Tunjic, C. "A Deep View-Point Language for Projective Modeling". In: *2017 IEEE 21st International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, 2017, pp. 133–142.

[ATM15] Atkinson, C., Tunjic, C., and Möller, T. "Fundamental Realization Strategies for Multi-view Specification Environments". In: *2015 IEEE 19th International Enterprise Distributed Object Computing Conference*. IEEE, 2015, pp. 40–49.

[AWY13] Azodolmolky, S., Wieder, P., and Yahyapour, R. "Cloud computing networking: challenges and opportunities for innovations". In: *IEEE Communications Magazine* 51.7 (2013), pp. 54–62.

[AZY13] Alnusair, A., Zhao, T., and Yan, G. "Rule-based detection of design patterns in program code". In: *International Journal on Software Tools for Technology Transfer* 16.3 (2013), pp. 315–334.

[Bac+00] Bachmann, F., Bass, L., Carriere, J., Clements, P., Garlan, D., Ivers, J., Nord, R., and Little, R. *Software Architecture Documentation in Practice: Documenting Architectural Layers*. 2000.

[Bac+63] Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., et al. "Revised report on the algorithmic language Algol 60". In: *Communications of the ACM* 6.1 (1963), pp. 1–17.

[Bac59] Backus, J. W. "The syntax and the semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference". In: *ICIP Proceedings*. 1959, pp. 125–132.

[Bal+06] Ballagas, R., Borchers, J., Rohs, M., and Sheridan, J. G. "The Smart Phone: A Ubiquitous Input Device". In: *IEEE Pervasive Computing* 5.1 (2006), pp. 70–77.

[Ban+93] Banker, R. D., Datar, S. M., Kemerer, C. F., and Zweig, D. "Software complexity and maintenance costs". In: *Communications of the ACM* 36.11 (1993), pp. 81–94.

[Bas92] Basili, V. R. "Software modeling and measurement: the Goal/Question/Metric paradigm". In: 1992.

[BB08] Battle, R. and Benson, E. "Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST)". In: *Journal of Web Semantics* 6.1 (2008). Semantic Web and Web 2.0, pp. 61–69.

[BCK22] Bass, L., Clements, P., and Kazman, R. *Software architecture in practice*. Fourth edition. Addison-Wesley, 2022. 438 pp. ISBN: 9780136885672.

[BCR94] Basili, V. R., Caldiera, G., and Rombach, H. D. "The Goal Question Metric Approach". In: *Encyclopedia of software engineering* (1994), pp. 528–532.

[BDS08] Benslimane, D., Dustdar, S., and Sheth, A. "Services Mashups: The New Generation of Web Applications". In: *IEEE Internet Computing* 12.5 (2008), pp. 13–15.

[BDS98] Banker, R. D., Davis, G. B., and Slaughter, S. A. "Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study". In: *Management Science* 44.4 (1998), pp. 433–450.

[Bec+10] Becker, S., Hauck, M., Trifu, M., Krogmann, K., and Kofroň, J. "Reverse Engineering Component Models for Quality Predictions". In: *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 194–197.

[Bec08] Becker, S. "Coupled model transformations for QoS enabled component-based software design". PhD thesis. 2008. 297 pp.

[Bec10] Becker, S. "The palladio component model. Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering; January 28 - 30, 2010, San José, CA, USA". In: *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*. Association for Computing Machinery, 2010, pp. 257–258.

[Bet16] Bettini, L. *Implementing domain-specific languages with Xtext and Xtend. Learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices*. Second edition. Packt Publishing, 2016. 1427 pp. ISBN: 9781786463272.

[Béz05] Bézivin, J. "On the unification power of models". In: *Software & Systems Modeling* 4.2 (2005), pp. 171–188.

[BHK11] Brosig, F., Huber, N., and Kounev, S. "Automated extraction of architecture-level performance models of distributed component-based systems". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 183–192.

[BKK09] Brosig, F., Kounev, S., and Krogmann, K. "Automated extraction of palladio component models from running enterprise Java applications". In: *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009.

[BKR07] Becker, S., Koziolek, H., and Reussner, R. "Model-Based performance prediction with the palladio component model". In: *Proceedings of the 6th International Workshop on Software and Performance*. Association for Computing Machinery, 2007, pp. 54–65.

[BKR09] Becker, S., Koziolek, H., and Reussner, R. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82.1 (2009). Special Issue: Software Performance - Modeling and Analysis, pp. 3–22.

[Bog+19] Bogner, J., Fritzsch, J., Wagner, S., and Zimmermann, A. "Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality". In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. Vol. 14. IEEE, 2019, pp. 187–195.

[Bor+08] Bork, M., Geiger, L., Schneider, C., and Zündorf, A. "Towards Roundtrip Engineering - A Template-Based Reverse Engineering Approach. 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings". In: *Model Driven Architecture – Foundations and Applications*. Springer Berlin Heidelberg, 2008, pp. 33–47.

[BR05] Blaha, M. and Rumbaugh, J. *Object-oriented modeling and design with UML*. 2. ed. Pearson/Prentice Hall, 2005. 477 pp. ISBN: 0130159204.

[Bra+08] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Online. 2008.

[Bre+18] Brecher, C., Kusmenko, E., Lindt, A., Rumpe, B., Storms, S., Wein, S., Wenckstern, M. von, and Wortmann, A. "Multi-Level Modeling Framework for Machine as a Service Applications Based on Product Process Resource Models". In: *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control*. Association for Computing Machinery, 2018.

[Bro+12] Brosch, F., Koziolek, H., Buhnova, B., and Reussner, R. "Architecture-Based Reliability Prediction with the Palladio Component Model". In: *IEEE Transactions on Software Engineering* 38.6 (2012), pp. 1319–1339.

[Bru+06] Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., and Sabetzadeh, M. "A manifesto for model merging". In: *Proceedings of the 2006 International Workshop on Global Integrated Model Management*. Association for Computing Machinery, 2006, pp. 5–12.

[Bru+10] Bruneliere, H., Cabot, J., Jouault, F., and Madiot, F. "MoDisco: a generic and extensible framework for model driven reverse engineering. Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, September 20 - 24, 2010, Antwerp, Belgium". In: *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, 2010, pp. 173–174.

[Bru+14] Brunelière, H., Cabot, J., Dupé, G., and Madiot, F. "MoDisco: A model driven reverse engineering framework". In: *Information and Software Technology* 56.8 (2014), pp. 1012–1032.

[Bru+15] Bruneliere, H., Perez, J. G., Wimmer, M., and Cabot, J. "EMF Views: A View Mechanism for Integrating Heterogeneous Models. 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings". In: *Conceptual Modeling*. v.9381. Springer International Publishing, 2015, pp. 317–325.

[BS22] Brahmaleen Kaur Sidhu, K. S. and Sharma, N. "A machine learning approach to software model refactoring". In: *International Journal of Computers and Applications* 44.2 (2022), pp. 166–177.

[Bur+14] Burger, E., Henss, J., Küster, M., Kruse, S., and Happe, L. "View-based model-driven software development with ModelJoin". In: *Software & Systems Modeling* 15.2 (2014), pp. 473–496.

[Bur14] Burger, E. "Flexible Views for View-based Model-driven Development". PhD thesis. 2014. 324 pp.

[BW84] Basili, V. R. and Weiss, D. M. "A Methodology for Collecting Valid Software Engineering Data". In: *IEEE Transactions on Software Engineering* SE-10.6 (1984), pp. 728–738.

[Cai+13] Cai, Y., Wang, H., Wong, S., and Wang, L. "Leveraging design rules to improve software architecture recovery. QoSA". In: *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*. Vol. 3. Association for Computing Machinery, 2013, pp. 133–142.

[Cam16] Campagne, F. *The MPS Language Workbench. Volume I*. Third edition, version 1.5.1, March 2016. Vol. 1. CreateSpace Independent Publishing, 2016. 250 pp. ISBN: 9781530533350.

[CC03] Chen, P. and Clothier, J. "Advancing systems engineering for systems-of-systems challenges". In: *Systems Engineering* 6.3 (2003), pp. 170–183.

[CD07] Canfora, G. and Di Penta, M. "New Frontiers of Reverse Engineering". In: *Future of Software Engineering (FOSE '07)*. IEEE, 2007, pp. 326–341.

[CDC11] Canfora, G., Di Penta, M., and Cerulo, L. "Achievements and challenges in software reverse engineering". In: *Commun. ACM* 54.4 (2011), pp. 142–151.

[CFM03] Clark, A. N., Futagami, T., and Mellor, S. J. "Guest Editors' Introduction: Model-Driven Development". In: *IEEE Software* 20.05 (2003), pp. 14–18.

[Chi+07] Chinnici, R., Moreau, J.-J., Ryman, A., and Weerawarana, S. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. Online. World Wide Web Consortium (W3C), 2007.

[CKK08] Chouambe, L., Klatt, B., and Krogmann, K. "Reverse Engineering Software-Models of Component-Based Systems". In: *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE, 2008, pp. 93–102.

[Cle14] *Documenting software architectures. Views and beyond*. 2. ed., 4. print. Literaturverz. S. 497 - 508. Addison-Wesley, 2014. 537 pp. ISBN: 9780321552686.

[Cle96] Clements, P. C. “A survey of architecture description languages”. In: *Proceedings of the 8th International Workshop on Software Specification and Design*. IEEE Comput. Soc. Press, 1996, pp. 16–25.

[Clo24] Cloud Native Computing Foundation (CNCF). *OpenTelemetry Specification*. Version 1.33.0. 2024. URL: https://github.com/open-telemetry/opentelemetry-specification.

[Cor+22] Cortellessa, V., Di Pompeo, D., Eramo, R., and Tucci, M. “A model-driven approach for continuous performance engineering in microservice-based systems”. In: *Journal of Systems and Software* 183 (2022), p. 111084.

[Cos+12] Cosentino, V., Cabot, J., Albert, P., Bauquel, P., and Perronnet, J. “A Model Driven Reverse Engineering Framework for Extracting Business Rules Out of a Java Application. 6th International Symposium, RuleML 2012, Montpellier, France, August 27-29, 2012. Proceedings”. In: *Rules on the Web: Research and Applications*. v.7438. Springer Berlin Heidelberg, 2012, pp. 17–31.

[Crn01] Crnkovic, I. “Component-based software engineering — new challenges in software development”. In: *Software Focus* 2.4 (2001), pp. 127–133.

[Crn03] Crnkovic, I. “Component-based software engineering - new challenges in software development”. In: *Proceedings of the 25th International Conference on Information Technology Interfaces, 2003. ITI 2003.* Univ. Zagreb, 2003, pp. 9–18.

[CS10] Cipresso, T. and Stamp, M. “Software Reverse Engineering”. In: *Handbook of Information and Communication Security*. Springer Berlin Heidelberg, 2010, pp. 659–696.

[Cur+02] Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., and Weerawarana, S. “Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI”. In: *IEEE Internet Computing* 6.2 (2002), pp. 86–93.

[CVE] Mitre Corporation. *Common Vulnerabilities and Exposures (CVE)*. 2024. URL: https://cwe.mitre.org.

[CVSS] Common Vulnerability Scoring System SIG. *Common Vulnerability Scoring System version 4.0: Specification Document*. 2023. URL: https://www.first.org/cvss/v4.0/specification-document.

[CWE] Mitre Corporation. *Common Weakness Enumeration (CWE)*. 2024. URL: https://cwe.mitre.org.

[Dan+24] Danial, A. et al. *AlDanial/cloc: v2.00*. 2024.

[Deh+22] Dehghani, M., Kolahdouz-Rahimi, S., Tisi, M., and Tamzalit, D. “Facilitating the migration to the microservice architecture via model-driven reverse engineering and reinforcement learning”. In: *Software and Systems Modeling* 21.3 (2022), pp. 1115–1133.

[Dek24] Deket, M. *mdeket/spring-cloud-movie-recommendation*. GitHub. original-date: 2016-11-29T13:02:38Z. 2024.

[Deu+04] Deursen, A. van, Hofmeister, C., Koschke, R., Moonen, L., and Riva, C. "Symphony: view-driven software architecture reconstruction". In: *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*. IEEE Comput. Soc, 2004, pp. 122–132.

[DGN05] Ducasse, S., Gîrba, T., and Nierstrasz, O. "Moose: an agile reengineering environment. Proceedings of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13); September 5 - 9, 2005, Lisbon, Portugal". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery, 2005, pp. 99–102.

[Di +02] Di Lucca, G. A., Fasolino, A. R., Pace, F., Tramontana, P., and De Carlini, U. "WARE: a tool for the reverse engineering of Web applications". In: *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*. IEEE Comput. Soc, 2002, pp. 241–250.

[Doc24] Docker, Inc. *Docker*. Version 26.1.3. Docker, Inc., 2024. URL: https://www.docker.com.

[DP09] Ducasse, S. and Pollet, D. "Software Architecture Reconstruction: A Process-Oriented Taxonomy". In: *IEEE Transactions on Software Engineering* 35.4 (2009), pp. 573–591.

[Dra+17] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. "Microservices: Yesterday, Today, and Tomorrow". In: *Present and Ulterior Software Engineering*. Springer International Publishing, 2017, pp. 195–216.

[Duc+17] Duc, A. N., Jabangwe, R., Paul, P., and Abrahamsson, P. "Security challenges in IoT development: a software engineering perspective". In: *Proceedings of the XP2017 Scientific Workshops*. Association for Computing Machinery, 2017.

[Ecl23] Eclipse Foundation. *Atlas Transformation Language (ATL)*. Version 4.9.0. Eclipse Foundation, 2023. URL: https://www.eclipse.org/atl/.

[Ecl24a] Eclipse Foundation. *Eclipse Integrated Development Environment (IDE)*. Version 4.27.0. Eclipse Foundation, 2024. URL: https://www.eclipse.org/.

[Ecl24b] Eclipse Foundation. *Eclipse Java Development Tools (JDT)*. Version 4.27.0. Eclipse Foundation, 2024. URL: https://www.eclipse.org/jdt.

[Ecl24c] Eclipse Foundation. *Eclipse Modeling Framework (EMF)*. Version 2.27. Eclipse Foundation, 2024. URL: https://www.eclipse.org/modeling/emf.

[Ecl24d] Eclipse Foundation. *Xtend*. Version 2.35.0. Eclipse Foundation, 2024. URL: https://www.eclipse.org/xtend/.

[ECMA] Ecma International. *ECMAScript*. 14th Edition. 2023. URL: https://ecma-international.org/publications-and-standards/standards/ecma-262/.

[Eis+18] Eismann, S., Walter, J., Kistowski, J. von, and Kounev, S. "Modeling of Parametric Dependencies for Performance Prediction of Component-Based Software Systems at Run-Time". In: *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018, pp. 135–13509.

[Eis+20] Eismann, S., Bezemer, C.-P., Shang, W., Okanović, D., and Hoorn, A. van. "Microservices: A Performance Tester's Dream or Nightmare?" In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ACM, 2020, pp. 138–149.

[Eld+24] Elder, S., Rahman, M. R., Fringer, G., Kapoor, K., and Williams, L. "A Survey on Software Vulnerability Exploitability Assessment". In: *ACM Comput. Surv.* 56.8 (2024), pp. 1–41.

[Far+19] Farias, K., Cavalcante, T., José Gonçales, L., and Bischoff, V. "UML2Merge: a UML extension for model merging". In: *IET Software* 13.6 (2019), pp. 575–586.

[Fav10] Favre, L. *Model Driven Architecture for Reverse Engineering Technologies: Strategic Directions and System Evolution. Strategic directions and system evolution.* IGI Global, 2010. 443 pp. ISBN: 9781615206506.

[FF95] Frakes, W. B. and Fox, C. J. "Sixteen questions about software reuse". In: *Commun. ACM* 38.6 (1995), 75–ff.

[Fin+92] Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., and Goedicke, M. "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development". In: *International Journal of Software Engineering and Knowledge Engineering* 02.01 (1992), pp. 31–57.

[FL14] Fowler, M. and Lewis, J. "Microservices: a definition of this new architectural term (2014)". In: *martinFowler.com* 1.1 (2014), pp. 1–1.

[FLP99] Fradet, P., Le Métayer, D., and Périn, M. "Consistency checking for multiple view software architectures". In: *ACM SIGSOFT Software Engineering Notes* 24.6 (1999), pp. 410–428.

[for23] fortiss. *CCE Trends*. 2023.

[FRS16] Fitriani, W. R., Rahayu, P., and Sensuse, D. I. "Challenges in agile software development: A systematic literature review". In: *2016 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*. Vol. 51. IEEE, 2016, pp. 155–164.

[Fur+16] Furdek, M., Wosinska, L., Goścień, R., Manousakis, K., Aibin, M., Walkowiak, K., Ristov, S., Gushev, M., and Marzo, J. L. "An overview of security challenges in communication networks". In: *2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM)*. Vol. 16. IEEE, 2016, pp. 43–50.

[Gar+11] Garcia, J., Popescu, D., Mattmann, C., Medvidovic, N., and Cai, Y. "Enhancing architectural recovery using concerns". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 552–555.

[Gar+14] Garzón, M. A., Lethbridge, T. C., Aljamaan, H., and Badreddin, O. "Reverse engineering of object-oriented code into Umple using an incremental and rule-based approach". In: *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., 2014, pp. 91–105.

[GBB12] Goldschmidt, T., Becker, S., and Burger, E. "Towards a tool-oriented taxonomy of view-based modelling. 14. - 16. März 2012, Bamberg". In: *Modellierung 2012*. GI e.V., 2012, pp. 59–74.

[GBJ02] Glinz, M., Berner, S., and Joos, S. "Object-oriented modeling with Adora". In: *Information Systems* 27.6 (2002), pp. 425–444.

[GGB23] Gueye, A., Galhardo, C. E. C., and Bojanova, I. "Critical Software Security Weaknesses". In: *IT Professional* 25.4 (2023), pp. 11–16.

[GIM13] Garcia, J., Ivkovic, I., and Medvidovic, N. "A comparative analysis of software architecture recovery techniques". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 486–496.

[Git24] GitHub. *GitHub Actions*. GitHub, 2024. URL: `https://github.com/features/actions`.

[Got+12] Gotel, O., Cleland-Huang, J., Hayes, J. H., Zisman, A., Egyed, A., Grünbacher, P., Dekhtyar, A., Antoniol, G., Maletic, J., and Mäder, P. "Traceability Fundamentals". In: *Software and Systems Traceability*. Springer London, 2012, pp. 3–22.

[GPD18] Guamán, D., Pérez, J., and Díaz, J. "Towards a (semi)-automatic reference process to support the reverse engineering and reconstruction of software architectures". In: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. Association for Computing Machinery, 2018.

[Gra+17] Granchelli, G., Cardarelli, M., Di Francesco, P., Malavolta, I., Iovino, L., and Di Salle, A. "MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-Based Systems". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 298–302.

[Gra24] Gradle Inc. *Gradle*. Version 8.7. Gradle Inc., 2024. URL: `https://gradle.org/`.

[Gro+21] Grohmann, J., Eismann, S., Bauer, A., Spinner, S., Blum, J., Herbst, N., and Kounev, S. "SARDE: A Framework for Continuous and Self-Adaptive Resource Demand Estimation". In: *ACM Transactions on Autonomous and Adaptive Systems* 15.2 (2021), pp. 1–31.

[Gst+24] Gstür, M., Kirschner, Y. R., Singh, S., and Koziolek, A. "MoCoRe - A Generic Model-Driven Composition and Rule-Based Refinement Framework". In: *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C), Hyderabad, 4th - 8th June 2024*. IEEE International Conference on Software Architecture Companion. ICSA-C 2024 (June 4–8, 2024). Institute of Electrical and Electronics Engineers (IEEE), 2024, pp. 273–280.

[GZ21] Genfer, P. and Zdun, U. "Identifying Domain-Based Cyclic Dependencies in Microservice APIs Using Source Code Detectors. 15th European Conference, ECSA 2021, Virtual Event, Sweden, September 13-17, 2021, Proceedings". In: *Software Architecture*. v.12857. Springer International Publishing, 2021, pp. 207–222.

[Hal+24] Hallvard, T. et al. *PlantUML in Eclipse*. Version 1.1.31. plantuml.com, 2024. URL: `https://github.com/hallvard/plantuml`.

[HB04] Haas, H. and Brown, A. *Web Services Glossary*. Tech. rep. W3C Working Group Note 11 February 2004. World Wide Web Consortium (W3C), 2004.

[HC01] *Component-based software engineering: putting the pieces together. Putting the pieces together*. 1. print. Addison-Wesley Longman Publishing Co., Inc., 2001. 818 pp. ISBN: 0201704854.

[Hea24] Headway Software Technologies Ltd. *Structure101*. Version 6. Headway Software Technologies Ltd., 2024. URL: `https://structure101.com/`.

[Heb+18] Hebig, R., Seidl, C., Berger, T., Pedersen, J. K., and Wąsowski, A. "Model transformation languages under a magnifying glass: a controlled experiment with Xtend, ATL, and QVT". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 2018, pp. 445–455.

[HKR11] Happe, J., Koziolek, H., and Reussner, R. "Facilitating Performance Predictions Using Software Components". In: *IEEE Software* 28.3 (2011), pp. 27–33.

[HKT22] Höppner, S., Kehrer, T., and Tichy, M. "Contrasting dedicated model transformation languages versus general purpose languages: a historical perspective on ATL versus Java based on complexity and size". In: *Softw. Syst. Model.* 21.2 (2022), pp. 805–837.

[HM+24] Hunter, J., McLaughlin, B., et al. *JDOM*. Version 2.0.6.1. JDOM Project, 2024. URL: `https://github.com/hunterhacker/jdom`.

[HNZ17] Haitzer, T., Navarro, E., and Zdun, U. "Reconciling software architecture and source code in support of software evolution". In: *Journal of Systems and Software* 123 (2017), pp. 119–144.

[Hoo+09] Hoorn, A. van, Rohr, M., Hasselbring, W., Waller, J., Ehlers, J., Frey, S., and Kieselhorst, D. "Continuous Monitoring of Software Services: Design and Application of the Kieker Framework". In: *Bericht des Instituts für Informatik*. Vol. 0921. Institut für Informatik, 2009, p. 27.

[Höp+22] Höppner, S., Haas, Y., Tichy, M., and Juhnke, K. "Advantages and disadvantages of (dedicated) model transformation languages: A qualitative interview study". In: *Empirical Software Engineering* 27.6 (2022).

[HSM92] Huang, H., Sugihara, K., and Miyamoto, I. "A rule-based tool for reverse engineering from source code to graphical models". In: *Proceedings Fourth International Conference on Software Engineering and Knowledge Engineering*. IEEE Comput. Soc. Press, 1992, pp. 178–185.

[HWH12] Hoorn, A. van, Waller, J., and Hasselbring, W. "Kieker: a framework for application performance monitoring and dynamic software analysis. Proceedings of the 3rd Joint WOSP/SIPEW International Conference on Peformance Engineering". In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, 2012, pp. 247–248.

[HZJ17] Heinrich, R., Zirkelbach, C., and Jung, R. "Architectural Runtime Modeling and Visualization for Quality-Aware DevOps in Cloud Applications". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 199–201.

[I42010] International Organization for Standardization. *ISO/IEC/IEEE 42010:2022(E) – Systems and software engineering – Architecture description*. 2nd ed. International Organization for Standardization, 2022.

[I62714] International Electrotechnical Commission. *IEC 62714-2:2022 — Engineering data exchange format for use in industrial automation systems engineering – Automation Markup Language*. International Electrotechnical Commission, 2022.

[Ian+23] Iannone, E., Guadagni, R., Ferrucci, F., De Lucia, A., and Palomba, F. "The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study". In: *IEEE Transactions on Software Engineering* 49.1 (2023), pp. 44–63.

[Ich+12] Ichii, M., Myojin, T., Nakagawa, Y., Chikahisa, M., and Ogawa, H. "A Rule-based Automated Approach for Extracting Models from Source Code". In: *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 308–317.

[Int24] International Business Machines Corporation (IBM). *IBM® Engineering Systems Design Rhapsody® (Rational Rhapsody)*. Version 10.0. International Business Machines Corporation (IBM), 2024. URL: `https://www.ibm.com/products/systems-design-rhapsody`.

[JAV09] Jansen, A., Avgeriou, P., and Ven, J. S. van der. "Enriching software architecture documentation". In: *Journal of Systems and Software* 82.8 (2009). SI: Architectural Decisions and Rationale, pp. 1232–1248.

[JAXRS] Eclipse Foundation. *Jakarta RESTful Web Services (JAX-RS) 4.0*. Version 4.0. 2024. URL: `https://jakarta.ee/specifications/restful-ws/4.0/`.

[JB05] Jansen, A. and Bosch, J. "Software Architecture as a Set of Architectural Design Decisions". In: *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*. IEEE, 2005, pp. 109–120.

[JK06] Jouault, F. and Kurtev, I. "Transforming Models with ATL. MoDELS 2005 International Workshop OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, Nfc, MDD, WUsCaM, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers". In: *Satellite Events at the MoDELS 2005 Conference*. Springer Berlin Heidelberg, 2006, pp. 128–138.

[JLS22] Jordan, S., Linsbauer, L., and Schaefer, I. "AutoArx: Digital Twins of Living Architectures. 16th European Conference, ECSA 2022, Prague, Czech Republic, September 19-23, 2022, Proceedings". In: *Software Architecture*. v.13444. Springer International Publishing, 2022, pp. 205–212.

[Jor+23] Jordan, S., König, C., Linsbauer, L., and Schaefer, I. "Automated Integration of Heteregeneous Architecture Information into a Unified Model. 17th European Conference, ECSA 2023, Istanbul, Turkey, September 18-22, 2023, Proceedings". In: *Software Architecture*. v.14212. Description based on publisher supplied metadata and other sources. Springer Nature Switzerland, 2023, pp. 83–99.

[JR01] Johnson, M. and Rosebrugh, R. "Engineering legacy information systems for internet based interoperation". In: *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*. IEEE Comput. Soc, 2001, pp. 32–39.

[JSQ24] JSQLParser Contributors. *JSqlParser*. Version 4.9. JSQLParser Project, 2024. URL: https://github.com/JSQLParser/JSqlParser.

[JT18] Joiner, K. F. and Tutty, M. G. "A tale of two allied defence departments: new assurance initiatives for managing increasing system complexity, interconnectedness and vulnerability". In: *Australian Journal of Multi-Disciplinary Engineering* 14.1 (2018), pp. 4–25.

[Kav14] Kavis, M. *Architecting The Cloud. Design decisions for cloud computing service models (SaaS, PaaS, and IaaS)*. John Wiley & Sons, Ltd, 2014, pp. 193–199. 11 pp. ISBN: 9781118691779.

[Kaz+94] Kazman, R., Bass, L., Abowd, G., and Webb, M. "SAAM: a method for analyzing the properties of software architectures". In: *Proceedings of 16th International Conference on Software Engineering*. Vol. 17. IEEE Comput. Soc. Press, 1994, pp. 81–90.

[KBH07] Koziolek, H., Becker, S., and Happe, J. "Predicting the Performance of Component-Based Software Architectures with Different Usage Profiles. Third International Conference on Quality of Software Architectures, QoSA 2007, Medford, MA, USA, July 11-13, 2007, Revised Selected Papers". In: *Software Architectures, Components, and Applications*. v.4880. Springer Berlin Heidelberg, 2007, pp. 145–163.

[KBL13] Kramer, M. E., Burger, E., and Langhammer, M. "View-centric engineering with synchronized heterogeneous models". In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. ACM, 2013.

[KBN12] Kahtan, H., Bakar, N. A., and Nordin, R. "Reviewing the challenges of security features in component based software development models". In: *2012 IEEE Symposium on E-Learning, E-Management and E-Services*. Vol. 1. IEEE, 2012, pp. 1–6.

[KBP20] Kroß, J., Bludau, P., and Pretschner, A. "Center for Code Excellence Trends – Eine Plattform für Trends in Softwaretechnologien". In: *Informatik Spektrum* 43.6 (2020), pp. 417–424.

[KC98] Kazman, R. and Carriere, S. J. "View extraction and view fusion in architectural understanding". In: *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*. IEEE Comput. Soc, 1998, pp. 290–299.

[KD17] König, H. and Diskin, Z. "Efficient Consistency Checking of Interrelated Models. 13th European Conference, ECMFA 2017, Held As Part of STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings". In: *Modelling Foundations and Applications*. v.10376. Springer International Publishing, 2017, pp. 161–178.

[KDM] Object Management Group (OMG). *OMG Knowledge Discovery Metamodel (OMG KDM)*. Version 1.4. 2016. URL: https://www.omg.org/spec/KDM/1.4/.

[Kei+24] Keim, J., Corallo, S., Fuchß, D., Hey, T., Telge, T., and Koziolek, A. "Recovering Trace Links Between Software Documentation And Code". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Association for Computing Machinery, 2024.

[Kir+23a] Kirschner, Y. R., Keim, J., Peter, N., and Koziolek, A. "Automated Reverse Engineering of the Technology-Induced Software System Structure. 17th European Conference, ECSA 2023, Istanbul, Turkey, September 18-22, 2023, Proceedings". In: *Software Architecture – 17th European Conference, ECSA 2023, Istanbul, Turkey, September 18–22, 2023, Proceedings. Ed.: B. Tekinerdogan*. 17th European Conference on Software Architecture. ECSA 2023 (Sept. 18–22, 2023). Vol. 14212. v.14212. 46.23.01; LK 01. Springer Nature Switzerland, 2023, pp. 283–291.

[Kir+23b] Kirschner, Y. R., Walter, M., Bossert, F., Heinrich, R., and Koziolek, A. "Automatic Derivation of Vulnerability Models for Software Architectures. IEEE International Conference on Software Architecture". In: *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C), L'Aquila, Italy, 13-17 March 2023*. 20th IEEE International Conference on Software Architecture. ICSA 2023 (Mar. 13–17, 2023). Vol. 33. 46.23.03; LK 01. Institute of Electrical and Electronics Engineers (IEEE), 2023, pp. 276–283.

[Kir+24a] Kirschner, Y. R., Gstür, M., Sağlam, T., Weber, S., and Koziolek, A. *Retriever: A View-Based Approach to Reverse Engineering Software Architecture Models*. Tech. rep. This is a preprint article, it offers immediate access but has not been peer reviewed. Elsevier B.V., 2024. 36 pp.

[Kir+24b] Kirschner, Y. R., Gstür, M., Sağlam, T., Weber, S., and Koziolek, A. "Retriever: A view-based approach to reverse engineering software architecture models". In: *Journal of Systems and Software (JSS)* (2024), p. 112277.

[Kir+24c] Kirschner, Y. R., Bossert, F., Gstür, M., Peter, N., Stahl, M., and Koziolek, A. *View-Based-Reverse-Engineering/Retriever: v5.2.0.202410181425*. Version v5.2.0.202410181425. 2024. DOI: 10.5281/zenodo.13951797. URL: https://github.com/View-Based-Reverse-Engineering/Retriever.

[Kir+24d] Kirschner, Y. R., Bossert, F., Stahl, M., and Koziolek, A. *View-Based-Reverse-Engineering/Retriever-Benchmark: v5.2.0.202410181425*. Version v5.2.0.202410181425. Zenodo, 2024. DOI: 10.5281/zenodo.13952589.

[Kir21] Kirschner, Y. R. "Model-Driven Reverse Engineering of Technology-Induced Architecture for Quality Prediction". In: *15th European Conference on Software Architecture - Companion (ECSA-C 2021), Virtual online (originally: Växjö, Sweden), September, 13-17, 2021. Ed.: R. Heinrich.* 15th European Conference on Software Architecture. ECSA 2021 (Sept. 13–17, 2021). Vol. 2978. CEUR-WS.org, 2021.

[Kis+18] Kistowski, J. von, Eismann, S., Schmitt, N., Bauer, A., Grohmann, J., and Kounev, S. "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2018.

[Kis+24] Kistowski, J. von, Eismann, S., Schmitt, N., Bauer, A., Grohmann, J., and other. *DescartesResearch/TeaStore*. GitHub. original-date: 2017-08-18T06:22:29Z. 2024.

[KKR10] Krogmann, K., Kuperberg, M., and Reussner, R. "Using Genetic Search for Reverse Engineering of Parametric Behavior Models for Performance Prediction". In: *IEEE Transactions on Software Engineering* 36.6 (2010), pp. 865–877.

[KKR11] Koziolek, A., Koziolek, H., and Reussner, R. "PerOpteryx: automated application of tactics in multi-objective software architecture optimization". In: *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS*. Association for Computing Machinery, 2011, pp. 33–42.

[Kla+21] Klare, H., Kramer, M. E., Langhammer, M., Werle, D., Burger, E., and Reussner, R. "Enabling consistency in view-based system development — The Vitruvius approach". In: *Journal of Systems and Software* 171 (2021), p. 110815.

[Kla22] Klare, H. "Building Transformation Networks for Consistent Evolution of Interrelated Models". PhD thesis. Karlsruher Institut für Technologie (KIT), 2022. 547 pp.

[Koz08] Koziolek, H. "Parameter dependencies for reusable performance specifications of software components". PhD thesis. 2008. 333 pp.

[Koz13] Koziolek, A. "Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes". PhD thesis. 2013. 555 pp.

[KR08a] Koziolek, H. and Reussner, R. "A Model Transformation from the Palladio Component Model to Layered Queueing Networks. SPEC International Performance Evaluation Workshop, SIPEW 2008, Darmstadt, Germany, June 27-28, 2008. Proceedings". In: *Performance Evaluation: Metrics, Models and Benchmarks*. Springer Berlin Heidelberg, 2008, pp. 58–78.

[KR08b] Krogmann, K. and Reussner, R. "Palladio – Prediction of Performance Properties. Comparing Software Component Models". In: *The Common Component Modeling Example: Comparing Software Component Models*. Springer Berlin Heidelberg, 2008, pp. 297–326.

[Kra19] Kramer, M. E. "Specification Languages for Preserving Consistency between Models of Different Languages". PhD thesis. Karlsruher Institut für Technologie (KIT), 2019. 440 pp.

[Kro+24] Krogmann, K., Bossert, F., Gstür, M., Wenzel, F., Rühle, M., and Kirschner, Y. R. *Service Effect Model Extractor*. Version 5.2.0. palladiosimulator.org, 2024. URL: `https://github.com/PalladioSimulator/Palladio-ReverseEngineering-SoMoX-SEFF`.

[Kro12] Krogmann, K. "Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis". PhD thesis. 2012. 371 pp.

[Kru95] Kruchten, P. B. "The 4+1 View Model of architecture". In: *IEEE Software* 12.6 (1995), pp. 42–50.

[KSV09] Klint, P., Storm, T. van der, and Vinju, J. "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation". In: *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2009, pp. 168–177.

[Kur08] Kurtev, I. "State of the Art of QVT: A Model Transformation Language Standard. Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers". In: *Applications of Graph Transformations with Industrial Relevance*. v.5088. Springer Berlin Heidelberg, 2008, pp. 377–393.

[Lan+16] Langhammer, M., Shahbazian, A., Medvidovic, N., and Reussner, R. H. "Automated Extraction of Rich Software Models from Limited System Information". In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, 2016, pp. 99–108.

[Lan+24] Lano, K., Haughton, H., Yuan, Z., and Alfraihi, H. "Agile model-driven reengineering". In: *Innovations in Systems and Software Engineering* (2024).

[Lan19] Langhammer, M. *Automated Coevolution of Source Code and Software Architecture Models*. 25. KIT Scientific Publishing, 2019, p. 376. 339 pp. ISBN: 978-3-7315-0783-3.

[Lat23] Lattix United States. *Lattix Architect*. Version 2023.1. Lattix United States, 2023. URL: https://www.lattix.com/.

[Lea+24] Leary, S. et al. *JSON in Java*. Version 20240303. json.org, 2024. URL: https://github.com/stleary/JSON-java.

[Leo+15] Leonhardt, S., Hettwer, B., Hoor, J., and Langhammer, M. "Integration of Existing Software Artifacts into a View- and Change-Driven Development Approach". In: *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-Based Software-Engineering*. Association for Computing Machinery, 2015, pp. 17–24.

[LFR13] Lehnert, S., Farooq, Q.-a., and Riebisch, M. "Rule-Based Impact Analysis for Heterogeneous Software Artifacts". In: *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 2013, pp. 209–218.

[Li+23] Li, K., Chen, S., Fan, L., Feng, R., Liu, H., Liu, C., Liu, Y., and Chen, Y. "Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 2023, pp. 921–933.

[LSF03] Lethbridge, T. C., Singer, J., and Forward, A. "How software engineers use documentation: the state of the practice". In: *IEEE Software* 20.6 (2003), pp. 35–39.

[Luk24] Lukyanchikov, A. *sqshq/piggymetrics*. GitHub. original-date: 2015-03-29T17:56:31Z. 2024.

[Lun08] Lungu, M. "Towards reverse engineering software ecosystems". In: *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 428–431.

[Mar+10] Martens, A., Koziolek, H., Becker, S., and Reussner, R. "Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering; January 28 - 30, 2010, San José, CA, USA". In: *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*. Association for Computing Machinery, 2010, pp. 105–116.

[Mar19] Martin, B. "Common Vulnerabilities Enumeration (CVE), Common Weakness Enumeration (CWE), and Common Quality Enumeration (CQE): Attempting to systematically catalog the safety and security challenges for modern, networked, software-intensive systems". In: *Ada Lett.* 38.2 (2019), pp. 9–42.

[McI68] McIlroy, M. D. "Mass-Produced Software Components". In: NATO Science Committee, 1968, pp. 88–98.

[Med+02] Medvidovic, N., Rosenblum, D. S., Redmiles, D. F., and Robbins, J. E. "Modeling software architectures in the Unified Modeling Language". In: *ACM Transactions on Software Engineering and Methodology* 11.1 (2002), pp. 2–57.

[MH16] Moura, J. and Hutchison, D. "Review and analysis of networking challenges in cloud computing". In: *Journal of Network and Computer Applications* 60 (2016), pp. 113–129.

[MK06] Mens, K. and Kellens, A. "IntensiVE, a toolsuite for documenting and checking structural source-code regularities". In: *Conference on Software Maintenance and Reengineering (CSMR'06)*. IEEE, 2006, pp. 10–248.

[MKW20] Meier, J., Kateule, R., and Winter, A. "Operator-based Viewpoint Definition. International Conference on Model-Driven Engineering and Software Development". In: *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development*. SciTePress, 2020, pp. 401–408.

[MM06] Mitchell, B. S. and Mancoridis, S. "On the automatic modularization of software systems using the Bunch tool". In: *IEEE Transactions on Software Engineering* 32.3 (2006), pp. 193–208.

[MNH15] Mahmood, S., Niazi, M., and Hussain, A. "Identifying the challenges for managing component-based development in global software development: Preliminary results". In: *2015 Science and Information Conference (SAI)*. Vol. 2. IEEE, 2015, pp. 933–938.

[Moo96] Moore, M. M. "Rule-based detection for reverse engineering user interfaces". In: *Proceedings of WCRE '96: 4rd Working Conference on Reverse Engineering*. IEEE Comput. Soc. Press, 1996, pp. 42–48.

[MOT18] Motta, R. C., Oliveira, K. M. de, and Travassos, G. H. "On challenges in engineering IoT software systems". In: *Proceedings of the XXXII Brazilian Symposium on Software Engineering*. Association for Computing Machinery, 2018, pp. 42–51.

[MRS17] Mushtaq, Z., Rasool, G., and Shehzad, B. "Multilingual Source Code Analysis: A Systematic Literature Review". In: *IEEE Access* 5 (2017), pp. 11307–11336.

[MT00] Medvidovic, N. and Taylor, R. N. "A classification and comparison framework for software architecture description languages". In: *IEEE Transactions on Software Engineering* 26.1 (2000), pp. 70–93.

[MT10] Medvidovic, N. and Taylor, R. N. "Software architecture: foundations, theory, and practice". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. Association for Computing Machinery, 2010, pp. 471–472.

[Mül+00] Müller, H. A., Jahnke, J. H., Smith, D. B., Storey, M.-A., Tilley, S. R., and Wong, K. "Reverse engineering: a roadmap. 2000; 22nd International Conference on Software Engineering". In: *Proceedings of the Conference on The Future of Software Engineering*. Association for Computing Machinery, 2000, pp. 47–60.

[Mül+18] Müller, R., Mahler, D., Hunger, M., Nerche, J., and Harrer, M. "Towards an Open Source Stack to Create a Unified Data Source for Software Analysis and Visualization". In: *2018 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 2018, pp. 107–111.

[Mur11] Murer, S. *Managed Evolution. A Strategy for Very Large Information Systems*. Springer-Verlag Berlin Heidelberg, 2011. 26468 pp. ISBN: 9783642016332.

[MWT95] Müller, H. A., Wong, K., and Tilley, S. R. "Understanding Software Systems Using Reverse Engineering Technology". In: *Object-Oriented Technology for Database and Software Systems*. WORLD SCIENTIFIC, 1995, pp. 240–252.

[Noa+22] Noaman, M., Khan, M. S., Abrar, M. F., Ali, S., Alvi, A., and Saleem, M. A. "Challenges in Integration of Heterogeneous Internet of Things". In: *Scientific Programming* 2022.1 (2022), pp. 1–14.

[Obj21] Object Management Group (OMG). *Common Object Request Broker Architecture (CORBA®)*. Version 3.4. 2021. URL: https://www.omg.org/spec/CORBA/3.4/.

[Oqu16] Oquendo, F. "Software Architecture Challenges and Emerging Research in Software-Intensive Systems-of-Systems. 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 – December 2, 2016, Proceedings". In: *Software Architecture*. v.9839. Springer International Publishing, 2016, pp. 3–21.

[Ora13] Oracle EJB 3.2 Expert Group. *JSR 345: Enterprise JavaBeans™, Version 3.2 EJB Core Contracts and Requirements*. Specification Lead: Marina Vatkina (Oracle). 2013. URL: https://jcp.org/aboutJava/communityprocess/final/jsr345/.

[Ora21] Oracle Corporation. *Project Nashorn*. Version 15.3. OpenJDK, 2021. URL: https://openjdk.org/projects/nashorn.

[Ora24] Oracle Corporation. *Java Properties*. Version 21. Oracle Corporation, 2024. URL: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Properties.html.

[OT19] Ozkan, N. and Tarhan, A. K. "Investigating Causes of Scalability Challenges in Agile Software Development from a Design Perspective". In: *2019 1st International Informatics and Software Engineering Conference (UBMYK)*. Vol. 4. IEEE, 2019, pp. 1–6.

[Pan10] Pandey, R. K. "Architectural description languages (ADLs) vs UML: a review". In: *ACM SIGSOFT Software Engineering Notes* 35.3 (2010), pp. 1–5.

[Par+24] Parsons, R., Laycock, R., Fowler, M., Subramaniam, B., Böckeler, B., Byars, B., Falconi Crispim, C., Doernenburg, E., Torre, F. de la, Xu, H., Lewis, J., Hoenig, M., Ormaza, M., Mason, M., Ford, N., Shah, P., Shaw, S., Natesan, S., Liu, S., Tania, S., Seth, V., and Amaral, W. *Technology Radar – An opinionated guide to today's technology landscape*. Tech. rep. 30. Thoughtworks, Inc., 2024. eprint: https://www.thoughtworks.com/content/dam/thoughtworks/documents/radar/2024/04/tr_technology_radar_vol_30_en.pdf.

[Pfe20] Pfeiffer, R.-H. "What constitutes Software? An Empirical, Descriptive Study of Artifacts. IEEE/ACM International Conference on Mining Software Repositories (MSR)". In: *Proceedings of the 17th International Conference on Mining Software Repositories*. Association for Computing Machinery, 2020, pp. 481–491.

[PGP11] Pérez-Castillo, R., Guzmán, I. G.-R. de, and Piattini, M. "Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems". In: *Computer Standards & Interfaces* 33.6 (2011), pp. 519–532.

[PH05] Puccinelli, D. and Haenggi, M. "Wireless sensor networks: applications and challenges of ubiquitous sensing". In: *IEEE Circuits and Systems Magazine* 5.3 (2005), pp. 19–31.

[PL03] Perrey, R. and Lycett, M. "Service-oriented architecture". In: *2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings.* IEEE Comput. Soc, 2003, pp. 116–119.

[PM04] Potter, B. and McGraw, G. "Software security testing". In: *IEEE Security & Privacy* 2.5 (2004), pp. 81–85.

[PW92] Perry, D. E. and Wolf, A. L. "Foundations for the study of software architecture". In: *SIGSOFT Softw. Eng. Notes* 17.4 (1992), pp. 40–52.

[QVT] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/-View/Transformation Specification.* Version 1.3. 2016. URL: https://www.omg.org/spec/QVT/1.3/.

[RAZ17] Raibulet, C., Arcelli Fontana, F., and Zanoni, M. "Model-Driven Reverse Engineering Approaches: A Systematic Literature Review". In: *IEEE Access* 5 (2017), pp. 14516–14542.

[RD99] Richner, T. and Ducasse, S. "Recovering high-level views of object-oriented applications from static and dynamic information". In: *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360).* IEEE, 1999, pp. 13–22.

[Rek85] Rekoff, M. G. "On reverse engineering". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-15.2 (1985), pp. 244–252.

[Reu+11] Reussner, R., Becker, S., Burger, E., Happe, J., Hauck, M., Koziolek, A., Koziolek, H., Krogmann, K., and Kuperberg, M. *The Palladio Component Model*. Tech. rep. 14. Karlsruher Institut für Technologie (KIT), 2011. 193 pp.

[Reu+16] *Modeling and simulating software architectures. The Palladio approach.* The MIT Press, 2016. 400 pp. ISBN: 9780262336789.

[RH08] Runeson, P. and Höst, M. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14.2 (2008), pp. 131–164.

[Ric06] Richards, R. "Representational State Transfer (REST)". In: *Pro PHP XML and Web Services.* Apress, 2006, pp. 633–672.

[Ric07] Richardson, L. *RESTful web services.* O'Reilly, 2007. ISBN: 0596529260.

[Ric20] Richards, M. *Fundamentals of software architecture. An engineering approach.* First edition. Unitary Architecture. O'Reilly, 2020. 1421 pp. ISBN: 9781492043423.

[RK09] Rathfelder, C. and Kounev, S. “Modeling event-driven service-oriented systems using the palladio component model”. In: *Proceedings of the 1st International Workshop on Quality of Service-Oriented Software Systems*. Association for Computing Machinery, 2009, pp. 33–38.

[RK11] Rathfelder, C. and Klatt, B. “Palladio Workbench: A Quality-Prediction Tool for Component-Based Architectures”. In: *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*. IEEE, 2011, pp. 347–350.

[ROC24] Rukmono, S. A., Ochoa, L., and Chaudron, M. “Deductive Software Architecture Recovery via Chain-of-thought Prompting”. In: *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. Association for Computing Machinery, 2024, pp. 92–96.

[Roq+24] Roques, A. et al. *PlantUML*. Version 1.2024.5. PlantUML, 2024. URL: `https://plantuml.com/`.

[RR02] Riva, C. and Rodriguez, J. V. “Combining static and dynamic views for architecture reconstruction”. In: *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*. IEEE Comput. Soc, 2002, pp. 47–55.

[RS04] Rugaber, S. and Stirewalt, K. “Model-driven reverse engineering”. In: *IEEE Software* 21.4 (2004), pp. 45–53.

[RS17] Reis, A. and Silva, A. R. da. “XIS-Reverse: A Model-driven Reverse Engineering Approach for Legacy Information Systems. International Conference on Model-Driven Engineering and Software Development”. In: *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development — Volume 1: MODELSWARD*. INSTICC. SciTePress, 2017, pp. 196–207.

[RSD+24] Rey, A., Szarliński, M., Dapeng, L., et al. *spring-petclinic/spring-petclinic-microservices*. GitHub. original-date: 2016-11-12T14:57:30Z. 2024.

[RT14] Reineke, J. and Tripakis, S. “Basic Problems in Multi-View Modeling. TACAS”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2014, pp. 217–232.

[Run+12] Runeson, P., Host, M., Rainer, A., and Regnell, B. *Case study research in software engineering. Guidelines and examples*. 1st ed. Includes bibliographical references and index. John Wiley & Sons, 2012. 1237 pp. ISBN: 9781118181003.

[Rya99] Ryan, H. W. “Managing Development in the Era of Large Complex Systems”. In: *Information Systems Management* 16.2 (1999), pp. 89–91.

[Sab+07] Sabetzadeh, M., Nejati, S., Easterbrook, S., and Chechik, M. “A Relationship-Driven Framework for Model Merging”. In: *International Workshop on Modeling in Software Engineering (MISE'07: ICSE Workshop 2007)*. IEEE, 2007, pp. 2–2.

[Sab+19] Sabir, U., Azam, F., Haq, S. U., Anwar, M. W., Butt, W. H., and Amjad, A. "A Model Driven Reverse Engineering Framework for Generating High Level UML Models From Java Source Code". In: *IEEE Access* 7 (2019), pp. 158931–158950.

[San+14] Sanaei, Z., Abolfazli, S., Gani, A., and Buyya, R. "Heterogeneity in Mobile Cloud Computing: Taxonomy and Open Challenges". In: *IEEE Communications Surveys & Tutorials* 16.1 (2014), pp. 369–392.

[SB12] Silva, L. de and Balasubramaniam, D. "Controlling software architecture erosion: A survey". In: *Journal of Systems and Software* 85.1 (2012). Dynamic Analysis and Testing of Embedded Software, pp. 132–151.

[SDG23] Sangaroonsilp, P., Dam, H. K., and Ghose, A. "On Privacy Weaknesses and Vulnerabilities in Software Systems". In: *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE Press, 2023, pp. 1071–1083.

[SHS09] Sarkar, V., Harrod, W., and Snavely, A. E. "Software challenges in extreme scale systems". In: *Journal of Physics: Conference Series* 180.1 (2009), p. 012045.

[Sil16] Sill, A. "The Design and Architecture of Microservices". In: *IEEE Cloud Computing* 3.5 (2016), pp. 76–80.

[SK03] Sendall, S. and Kozaczynski, W. "Model transformation: the heart and soul of model-driven software development". In: *IEEE Software* 20.5 (2003), pp. 42–45.

[SK19] Schneider, Y. R. and Koziolek, A. "Towards Reverse Engineering for Component-Based Systems with Domain Knowledge of the Technologies Used". In: *Proceedings of the 10th Symposium on Software Performance (SSP)*. 10th Symposium on Software Performance. SSP 2019 (Nov. 4–6, 2019). GI, 2019, pp. 35–37.

[SK24] Singh, S. and Koziolek, A. "Automated Reverse Engineering for MoM-Based Microservices (ARE4MOM) Using Static Analysis". In: *2024 IEEE 21st International Conference on Software Architecture (ICSA)*. IEEE, 2024, pp. 12–22.

[SM23] Santiago, N. and Mendez, J. "Analysis of Common Vulnerabilities and Exposures to Produce Security Trends". In: *Proceedings of the 2022 International Conference on Cyber Security*. Association for Computing Machinery, 2023, pp. 16–19.

[Sny22] Snyk Ltd. *Remote Code Execution: Affecting org.springframework: spring-beans package, versions [,5.2.20) [5.3.0, 5.3.18)*. 2022. URL: https://security.snyk.io/vuln/SNYK-JAVA-ORGSPRINGFRAMEWORK-2436751.

[Sny24] Snyk Ltd. *Snyk CLI*. Version 1.1291.1. Snyk Ltd., 2024. URL: https://docs.snyk.io/snyk-cli.

[Som18] Sommerville, I. *Software Engineering*. 10., aktualisierte Auflage. Pearson, 2018. 896 pp. ISBN: 9783868943443.

[SPDX] Linux Foundation. *System Package Data Exchange (SPDX)*. Version 3.0.0. 2024. URL: https://spdx.github.io/spdx-spec/v3.0/.

[SSG13] Sánchez Ramón, Ó., Sánchez Cuadrado, J., and García Molina, J. "Model-driven reverse engineering of legacy graphical user interfaces". In: *Automated Software Engineering* 21.2 (2013), pp. 147–186.

[Sta23] Stack Overflow. *Stack Overflow Developer Survey 2023*. 2023.

[Sta73] Stachowiak, H. *Allgemeine Modelltheorie*. Springer, 1973. 494 pp. ISBN: 0387811060.

[Ste11] *Eclipse modeling framework. EMF*. 2. ed., rev. and updated, 2. printing. Addison-Wesley, 2011. 704 pp. ISBN: 0321331885.

[Ste19] Stevens, P. "Maintaining consistency in networks of models: bidirectional transformations in the large". In: *Software and Systems Modeling* 19.1 (2019), pp. 39–65.

[Str+06] Stringfellow, C., Amory, C. D., Potnuri, D., Andrews, A., and Georg, M. "Comparison of software architecture reverse engineering methods". In: *Information and Software Technology* 48.7 (2006), pp. 484–497.

[STV18] Soldani, J., Tamburri, D. A., and Van Den Heuvel, W.-J. "The pains and gains of microservices: A Systematic grey literature review". In: *Journal of Systems and Software* 146 (2018), pp. 215–232.

[Sus+23] Sushma, D., Nalini, M. K., Ashok Kumar, R., and Nidugala, M. "To Detect and Mitigate the Risk in Continuous Integration and Continues Deployments (CI/CD) Pipelines in Supply Chain Using Snyk tool". In: *2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*. IEEE, 2023, pp. 1–10.

[SVL08] Sangwan, R. S., Vercellone-Smith, P., and Laplante, P. A. "Structural Epochs in the Complexity of Software over Time". In: *IEEE Software* 25.4 (2008), pp. 66–73.

[SWC93] Selfridge, P. G., Waters, R. C., and Chikofsky, E. J. "Challenges to the field of reverse engineering". In: *[1993] Proceedings Working Conference on Reverse Engineering*. Vol. 18. IEEE Comput. Soc. Press, 1993, pp. 144–150.

[SWK22] Singh, S., Werle, D., and Koziolek, A. "ARCHI4MOM: Using Tracing Information to Extract the Architecture of Microservice-Based Systems from Message-Oriented Middleware. 16th European Conference, ECSA 2022, Prague, Czech Republic, September 19-23, 2022, Proceedings". In: *Software Architecture*. v.13444. Springer International Publishing, 2022, pp. 189–204.

[SysML] Object Management Group (OMG). *OMG System Modeling Language (OMG SysML)*. Version 1.6. 2019. URL: https://www.omg.org/spec/SysML/1.6/.

[Szy98] Szyperski, C. *Component software: beyond object-oriented programming. Beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., 1998. 411 pp. ISBN: 0201178885.

[Ter19] Terzo, O. *Heterogeneous Computing Architectures: Challenges and Vision. Challenges and vision*. 1st ed. Taylor & Francis, 2019. 1 p. ISBN: 9780429399602.

[TH00] Tzerpos, V. and Holt, R. C. “ACCD: an algorithm for comprehension-driven clustering”. In: *Proceedings Seventh Working Conference on Reverse Engineering*. IEEE Comput. Soc, 2000, pp. 258–267.

[The20] The OSGi Alliance. *OSGi Core*. Release 8. 2020. URL: https://docs.osgi.org/specification/osgi.core/8.0.0.

[Thö15] Thönes, J. “Microservices”. In: *IEEE Software* 32.1 (2015), pp. 116–116.

[Tho24] Thoughtworks Holding, Inc. *Technology Radar*. 2024.

[Ton+07] Tonella, P., Torchiano, M., Du Bois, B., and Systä, T. “Empirical studies in reverse engineering: state of the art and future trends”. In: *Empirical Software Engineering* 12.5 (2007), pp. 551–571.

[TR02] Tonella, P. and Ricca, F. “Dynamic model extraction and statistical analysis of Web applications”. In: *Proceedings. Fourth International Workshop on Web Site Evolution*. IEEE Comput. Soc, 2002, pp. 43–52.

[TS24] Tollefson, D. and Spyker, A. *acmeair/acmeair*. GitHub. original-date: 2013-05-28T20:44:33Z. 2024.

[TZD08] Tran, H., Zdun, U., and Dustdar, S. “View-Based Reverse Engineering Approach for Enhancing Model Interoperability and Reusability in Process-Driven SOAs. 10th International Conference on Software Reuse, ICSR 2008, Bejing, China, May 25-29 2008”. In: *High Confidence Software Reuse in Large Systems*. v.5030. Springer Berlin Heidelberg, 2008, pp. 233–244.

[UML] Object Management Group (OMG). *OMG Unified Modeling Language (OMG UML)*. Version 2.5.1. 2017. URL: https://www.omg.org/spec/UML/2.5.1/.

[US 23] U.S. National Vulnerability Database (NVD). *CVE-2022-22965 Detail*. 2023. URL: https://nvd.nist.gov/vuln/detail/cve-2022-22965.

[US 24] U.S. government. *U.S. National Vulnerability Database (NVD)*. U.S. National Institute of Standards and Technology (NIST). 2024. URL: https://nvd.nist.gov.

[Vit+04] Vittorini, V., Iacono, M., Mazzocca, N., and Franceschinis, G. “The OsMoSys approach to multi-formalism modeling of systems”. In: *Software & Systems Modeling* 3.1 (2004), pp. 68–81.

[VMw22] VMware LLC. *CVE-2022-22965: Spring Framework RCE via Data Binding on JDK 9+*. 2022. URL: https://spring.io/security/cve-2022-22965.

[VMw24] VMware LLC. *Spring Framework*. Version 6.1.8. VMware LLC, 2024. URL: https://spring.io/projects/spring-framework.

[Voe+13] Voelter, M., Ratiu, D., Kolb, B., and Schaetz, B. “mbeddr: instantiating a language workbench in the embedded software domain”. In: *Automated Software Engineering* 20.3 (2013), pp. 339–390.

[Völ13] *DSL engineering. Designing, implementing and using domain-specific languages*. CreateSpace Independent Publishing Platform, 2013. 558 pp. ISBN: 1481218581.

[WAA85] Wood-Harper, A. T., Antill, L., and Avison, D. E. *Information systems definition: the Multiview approach*. Blackwell Scientific Publications, Ltd., 1985.

[Wal+17] Walter, J., Stier, C., Koziolek, H., and Kounev, S. "An Expandable Extraction Framework for Architectural Performance Models". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. Association for Computing Machinery, 2017, pp. 165–170.

[Wal19] Walter, J. C. "Automation in Software Performance Engineering Based on a Declarative Specification of Concerns". doctoralthesis. Universität Würzburg, 2019.

[Wal24] Walter, M. "Context-based Access Control and Attack Modelling and Analysis". PhD thesis. Karlsruher Institut für Technologie (KIT), 2024. 309 pp.

[Wan+04] Wang, H., Huang, J. Z., Qu, Y., and Xie, J. "Web services: problems and future directions". In: *Journal of Web Semantics* 1.3 (2004), pp. 309–320.

[Wan+23] Wan, Z., Zhang, Y., Xia, X., Jiang, Y., and Lo, D. "Software Architecture in Practice: Challenges and Opportunities". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 2023, pp. 1457–1469.

[WHR22] Walter, M., Heinrich, R., and Reussner, R. "Architectural Attack Propagation Analysis for Identifying Confidentiality Issues". In: *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. IEEE, 2022, pp. 1–12.

[WHR23] Walter, M., Heinrich, R., and Reussner, R. "Architecture-Based Attack Path Analysis for Identifying Potential Security Incidents. European Conference on Software Architecture". In: *Software Architecture*. 14212. Literaturangaben. Springer Nature Switzerland, 2023, pp. 37–53.

[Wit24] Wittberger, G. *georgwittberger/apache-spring-boot-microservice-example*. GitHub. original-date: 2017-04-17T15:14:04Z. 2024.

[WLS20] Waseem, M., Liang, P., and Shahin, M. "A Systematic Mapping Study on Microservices Architecture in DevOps". In: *Journal of Systems and Software* 170 (2020), p. 110798.

[Wol24] Wolff, E. *ewolff/microservice*. GitHub. original-date: 2015-04-01T21:36:42Z. 2024.

[WWA19] Werner, C., Wimmer, M., and Aßmann, U. "A Generic Language for Query and Viewtype Generation By-Example. International Conference on Model-Driven Engineering and Software Development". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. SCITEPRESS - Science and Technology Publications, Lda., 2019, pp. 379–386.

[Yan+21] Yang, Z., Qiu, Z., Zhou, Y., Huang, Z., Bodeveix, J.-P., and Filali, M. "C2AADL_-Reverse: A model-driven reverse engineering approach to development and verification of safety-critical software". In: *Journal of Systems Architecture* 118 (2021), p. 102202.

[YP21] Yazdi, M. A. and Politze, M. "Reverse Engineering: The University Distributed Services. 2020". In: *Proceedings of the Future Technologies Conference (FTC) 2020, Volume 2*. Vol. Volume 2. 1289. Springer International Publishing, 2021, pp. 223–238.

[YX16] Yang, R. and Xu, J. "Computing at Massive Scale: Scalability and Dependability Challenges". In: *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. Vol. 139. IEEE, 2016, pp. 386–397.

[ZB12] Zhang, C. and Budgen, D. "What Do We Know about the Effectiveness of Software Design Patterns?" In: *IEEE Transactions on Software Engineering* 38.5 (2012), pp. 1213–1231.

Declaration of Generative AI and AI-Assisted Technologies in the Writing Process

While preparing this thesis, we used Grammarly[1], LanguageTool[2], ChatGPT[3], and DeepL[4] Translator and Write to recommend and insert replacement text, perform writing style, spelling, and grammar checks and corrections, and do text translations and abridgments. After using these tools and services, we reviewed and edited the content as needed. We take full responsibility for the content of the thesis.

[1] https://grammarly.com
[2] https://languagetool.org
[3] https://chatgpt.com
[4] https://deepl.com

Appendix

Reproduction Package

We have comprehensively described the implementation of our Retriever approach and made the relevant artifacts available on GitHub and Zenodo [Kir+24d; Kir+24c]. The evaluation results can be reproduced using the case studies in these repositories. Our approach was evaluated using version *v5.2.0.202410181425*.

Note that the implementation of our approach and these case studies are evolving, so we have added the semantic version number and date of last use for the evaluation. In addition, we provide a package on GitHub and Zenodo [Kir+24d; Kir+24c] with all the artifacts and a reproducible environment. The reproduction package includes the presented artifact versions and a bundled execution environment to facilitate their setup.

For each case study, Table A.1 links to the configuration of our Retriever approach, the automatically generated architectural model, and a final evaluation report of our approach. In Listings A.1, A.2, A.3, A.4, A.5, A.6, A.7 and A.8, the case study, in which a gold standard for component-based architecture was created manually, is accompanied by the corresponding PlantUML descriptions.

Version	Results	Configuration	Report
	`acmeair/acmeair`		
f161227	PCM	Configuration	Report
	`anilallewar/microservices-basics-spring-boot`		
ac0a249	PCM	Configuration	Report
	`apssouza22/java-microservice`		
9172a90	PCM	Configuration	Report
	`callistaenterprise/blog-microservices`		
1681bdb	PCM	Configuration	Report
	`cloudscale-project/cloudstore`		
dcbf23a	PCM	Configuration	Report
	`corona-warn-app`		
bbc28be	PCM	Configuration	Report
	`corona-warn-app/cwa-dcc-server`		
71f5933	PCM	Configuration	Report
	`corona-warn-app/cwa-log-upload`		
4d9b378	PCM	Configuration	Report

Version	Results	Configuration	Report
	corona-warn-app/cwa-ppa-server		
d441e87	PCM	Configuration	Report
	corona-warn-app/cwa-server		
c61d55f	PCM	Configuration	Report
	corona-warn-app/cwa-testresult-server		
d2068cc	PCM	Configuration	Report
	corona-warn-app/cwa-verification-portal		
e9b30bc	PCM	Configuration	Report
	corona-warn-app/cwa-verification-server		
bbc28be	PCM	Configuration	Report
	descartesresearch/teastore		
5da4ada	PCM	Configuration	Report
	ewolff/microservice		
d1ccf50	PCM	Configuration	Report
	fernandoabcampos/spring-netflix-oss-microservices		
8668c4d	PCM	Configuration	Report
	fudanselab/train-ticket		
313886e	PCM	Configuration	Report
	georgwittberger/apache-spring-boot-microservice-example		
5b97bd9	PCM	Configuration	Report
	jferrater/tap-and-eat-microservices		
3ad20b8	PCM	Configuration	Report
	kbastani/spring-cloud-event-sourcing-example		
01f435d	PCM	Configuration	Report
	kit-recipe-app/recipebackendnew		
c4754f0	PCM	Configuration	Report
	kit-sdq/esda		
886c1af	PCM	Configuration	Report
	koushikkothagal/spring-boot-microservices-workshop		
8b01c6d	PCM	Configuration	Report
	mdeket/spring-cloud-movie-recommendation		
5aa5ee9	PCM	Configuration	Report
	meet-eat/meet-eat-server		
9aca899	PCM	Configuration	Report

Version	Results	Configuration	Report
	`openmrs/openmrs-core`		
61a27fa	PCM	Configuration	Report
	`openmrs/openmrs-module-webservices.rest`		
61a27fa	PCM	Configuration	Report
	`piomin/sample-spring-oauth2-microservices`		
05f390e	PCM	Configuration	Report
	`rohitghatol/spring-boot-microservices`		
a3c9df9	PCM	Configuration	Report
	`shabbirdwd53/springboot-microservice`		
e0faba5	PCM	Configuration	Report
	`spring-petclinic/spring-petclinic-microservices`		
06a9bac	PCM	Configuration	Report
	`sqshq/piggymetrics`		
6bb2cf9	PCM	Configuration	Report
	`webgoat/webgoat`		
bb6e84d	PCM	Configuration	Report
	`yidongnan/spring-cloud-netflix-example`		
3b86bf0	PCM	Configuration	Report

Table A.1.: The table provides a comprehensive overview of the case studies, accompanied by hyperlinks to the exact version, results, configuration, and report. The hyperlinks are direct to other online sources for the above items and are accessible from within the reproduction package.

```
@startuml https://github.com/acmeair/acmeair

component "BookingService" {
   [BookingsREST]
   portin " " as Booking
   "Booking" -- [BookingsREST]
   portout " " as pBooking
   [BookingsREST] ..> "pBooking"
}

interface "iBooking" as iBooking
iBooking -- "Booking"

component "CustomerService" {
   [CustomerLoader]
   [CustomerREST]
   [KeyGenerator]
   [LoginREST]
```

```
19     portin " " as Customer
20     portin " " as Login
21     [CustomerREST] ..> [CustomerLoader] : requires
22     [CustomerREST] ..> [KeyGenerator] : requires
23     [LoginREST] ..> [CustomerLoader] : requires
24     "Customer" -- [CustomerREST]
25     "Login" -- [LoginREST]
26     portout " " as pCustomerLoader
27     [CustomerLoader] ..> "pCustomerLoader"
28 }
29
30 interface "iCustomer" as iCustomer
31 interface "iLogin" as iLogin
32 iCustomer -- "Customer"
33 iLogin -- "Login"
34
35 component "FlightService" {
36     [FlightLoader]
37     [FlightREST]
38     portin " " as Flight
39     [FlightREST] ..> [FlightLoader] : requires
40     "Flight" -- [FlightREST]
41     portout " " as pFlightLoader
42     [FlightLoader] ..> "pFlightLoader"
43 }
44
45 interface "iFlight" as iFlight
46 iFlight -- "Flight"
47
48 component "TransactionService" {
49     [RESTCookieSessionFilter]
50     [WXSSessionManager]
51     portin " " as Transaction
52     portout " " as pCustomer
53     [RESTCookieSessionFilter] ..> "pCustomer"
54     [RESTCookieSessionFilter] ..> [WXSSessionManager] : requires
55     "Transaction" -- [RESTCookieSessionFilter]
56     portout " " as pTransaction
57     [WXSSessionManager] ..> "pTransaction"
58 }
59
60 "pCustomer" ..> [iCustomer] : requires
61 interface "iTransaction" as iTransaction
62 iTransaction -- "Transaction"
63
64 [WebApp]
65 [WebApp] ..> "iBooking" : requires
```

```
[WebApp] ..> "iCustomer" : requires
[WebApp] ..> "iFlight" : requires
[WebApp] ..> "iLogin" : requires
[WebApp] ..> "iTransaction" : requires

[DataService]
"pBooking" ..> [DataService] : requires
"pCustomerLoader" ..> [DataService] : requires
"pFlightLoader" ..> [DataService] : requires
"pTransaction" ..> [DataService] : requires

@enduml
```

Listing A.1: Our gold standard for the component-based architecture of the Acme Air case study. The representation of this software architecture is based on PlantUML notation.

```
@startuml https://github.com/anilallewar/microservices-basics-spring-boot
top to bottom direction

component [api-gateway] as gateway

component [zipkin-server] as zipkin

component [webservice-registry] as registry

component [config-server] as config

component [web-portal] as portal

interface "/task-service" as generalInterface

interface "/user-service" as generalInterface1

interface "/comments" as CommentsInterface

component [task-webservice] as task {
   portin " " as TaskIn1

   component [CommentsService] as CommentsService
   component [TaskController] as TaskController
   portout " " as TaskOut1

   generalInterface -- TaskIn1
   TaskIn1 -- TaskController
   TaskController ..> CommentsService : requires
   CommentsService .. TaskOut1
   TaskOut1 ..> CommentsInterface : requires
}
```

```
33
34 component [comments-webservice] as comments {
35    portin " " as commentsIn1
36    component [CommentsController] as CommentsController
37
38    CommentsInterface -- commentsIn1
39    commentsIn1 -- CommentsController
40 }
41
42 component [user-webservice] as user {
43    portin " " as UserIn1
44    component [UserController] as UserController
45
46    generalInterface1 -- UserIn1
47    UserIn1 -- UserController
48 }
49
50 interface "/userauth" as authInterface
51
52 component [auth-server] as auth {
53    portin " " as authIn
54    component [AuthUserController] as AuthUserController
55    authInterface -- authIn
56    authIn -- AuthUserController
57 }
58
59 gateway ..> generalInterface : depends on
60 gateway ..> generalInterface1 : depends on
61 gateway ..> CommentsInterface : depends on
62 gateway ..> authInterface : depends on
63
64 comments ..> config : use
65 auth ..> config : use
66 gateway ..> config : use
67 task ..> config : use
68 user ..> config : use
69 portal ..> config : use
70 registry ..> config : use
71 zipkin ..> config : use
72
73 comments ..> registry : use
74 auth ..> registry : use
75 gateway ..> registry : use
76 task ..> registry : use
77 user ..> registry : use
78 portal ..> registry : use
79
```

```
comments ..> zipkin : Sends Tracing Data
task ..> zipkin : Sends Tracing Data
user ..> zipkin : Sends Tracing Data

@enduml
```

Listing A.2: Our gold standard for the component-based architecture of the Spring Boot Microservices Template case study. The representation of this software architecture is based on PlantUML notation.

```
@startuml https://github.com/DescartesResearch/TeaStore
skinparam fixCircleLabelOverlapping true
skinparam componentStyle uml2

package "Frontend" {
   component [Web UI]
   interface "Web UI" as interface.WebUI
   [Web UI] -- interface.WebUI
}

package "Database" {
   component [MariaDB]
   interface "Persistence" as interface.Persistence
   [MariaDB] -- interface.Persistence
}

package "Services" {
   component [Authentication Service]
   interface "Shopping Cart\nAuthentication" as interface.AuthCart

   component [Registry Service]

   component [Image Service]
   [Registry Service]..>[Image Service] : requires

   component [Persistence Service]
   [Registry Service]..>[Persistence Service] : requires

   component [Recommendation Service]
   [Registry Service]..>[Recommendation Service] : requires

}

interface "Load Balancer" as interface.LoadBalancer
[Registry Service] -- interface.LoadBalancer
[Authentication Service] ..> interface.LoadBalancer : requires
[Web UI] ..> interface.LoadBalancer : requires

[Authentication Service] -- interface.AuthCart
```

```
[Registry Service]..>interface.AuthCart : requires

[Persistence Service]..>interface.Persistence : requires

@enduml
```

Listing A.3: Our gold standard for the component-based architecture of the Tea Store case study. The representation of this software architecture is based on PlantUML notation.

```
@startuml https://github.com/ewolff/microservice

interface "/catalog" as catalogInterface
component [catalog] as catalog{
   portin " " as catalogIn
   component [CatalogController] as CatalogController
   component [ItemRepository] as ItemRepository
   component [CatalogApp] as CatalogApp

   catalogInterface -- catalogIn
   catalogIn -- CatalogController
   CatalogController ..> ItemRepository : requires
   CatalogApp ..> ItemRepository : requires
}

interface "/customer" as customerInterface
component [customer] as customer {
   portin " " as customerIn
   component [CustomerController] as CustomerController
   component [CustomerRepository] as CustomerRepository
   component [CustomerApp] as CustomerApp

   customerInterface --customerIn
   customerIn -- CustomerController
   CustomerController ..> CustomerRepository : requires
   CustomerApp ..> CustomerRepository : requires
}

interface "/order" as orderInterface
component [order] as order {
   portin " " as orderIn
   component [OrderController] as OrderController
   component [OrderRepository] as OrderRepository
   component [OrderService] as OrderService
   component [CustomerClient] as CustomerClient
   component [CatalogClient] as CatalogClient
   portout " " as orderCustomer
   portout " " as orderCatalog

```

```
40     orderInterface -- orderIn
41     orderIn -- OrderController
42     OrderController ..> OrderRepository : requires
43     OrderController ..> OrderService : requires
44     OrderService ..> OrderRepository : requires
45     OrderService ..> CustomerClient : requires
46     OrderService ..> CatalogClient : requires
47     OrderController ..> CustomerClient : requires
48     OrderController ..> CatalogClient : requires
49     CustomerClient .. orderCustomer
50     orderCustomer ..> customerInterface : requires
51     CatalogClient .. orderCatalog
52     orderCatalog ..> catalogInterface : requires
53  }
54
55  component [eureka] as eureka
56
57  interface "/api" as entryPoint
58  component [zuul] as zuul
59
60  entryPoint -- zuul
61  zuul ..> orderInterface : depends on
62  zuul ..> catalogInterface : depends on
63  zuul ..> customerInterface : depends on
64
65  order ..> eureka : use
66  catalog ..> eureka : use
67  customer ..> eureka : use
68  zuul ..> eureka : use
69
70  @enduml
```

Listing A.4: Our gold standard for the component-based architecture of the Microservice Sample case study. The representation of this software architecture is based on PlantUML notation.

```
1  @startuml ⟷
       https://github.com/georgwittberger/apache-spring-boot-microservice-example
2
3  component [web-server] as web
4
5  interface "/cart" as cartInterface
6  component [cart-service] as cart {
7     portin " " as cartIn
8     component [CartController] as CartController
9     component [CartService] as CartService
10
11    cartInterface -- cartIn
12    cartIn -- CartController
```

```
13     CartController ..> CartService : requires
14  }
15
16  interface "/product" as productInterface
17  component [product-service] as product{
18     portin " " as productIn
19     component [ProductController] as ProductController
20     component [ProductService] as ProductService
21
22     productInterface -- productIn
23     productIn -- ProductController
24     ProductController ..> ProductService : requires
25  }
26
27  interface "/" as contenInterface
28  component [content-service] as content {
29     portin " " as contentIn
30     component [ContentController] as ContentController
31
32     contenInterface -- contentIn
33     contentIn -- ContentController
34  }
35
36  web ..> cartInterface : forward
37  web ..> productInterface : forward
38  web ..> contenInterface : forward
39
40  @enduml
```

Listing A.5: Our gold standard for the component-based architecture of the Apache Web Server and Spring Boot Microservice Example case study. The representation of this software architecture is based on PlantUML notation.

```
1   @startuml https://github.com/mdeket/spring-cloud-movie-recommendation
2
3   interface "/movie" as MovieInterface1
4
5   component [movie-service] as movieService{
6      portin " " as movieIn1
7      component [MovieController] as MovieController
8      component [MovieRepo] as MovieRepo1
9
10     MovieInterface1 -- movieIn1
11
12     MovieController ..> MovieRepo1 : requires
13     movieIn1 -- MovieController
14  }
15
16  interface "/recommendation" as recoInterface1
```

```
17
18 component [recommendation-service] as recoService {
19    portin " " as RecoIn
20    component [RecommendationController] as RecoController
21    component [UserRepo] as UserRepo2
22    component [MovieRepo] as MovieRepo2
23
24    recoInterface1 -- RecoIn
25    RecoIn -- RecoController
26    RecoController ..> UserRepo2 : requires
27    RecoController ..> MovieRepo2 : requires
28 }
29
30 interface "/user" as userInterface1
31
32 component [user-service] as userService {
33    portin " " as UserIn1
34
35    component [UserController] as UserController
36    component [UserRepo] as UserRepo1
37
38    userInterface1 -- UserIn1
39
40    UserIn1 -- UserController
41
42    UserController ..> UserRepo1:requires
43 }
44
45 interface "/api" as ClientInterface
46
47 component [recommendation-client] as recoClient{
48    portin " " as ClientIn
49    component [MainController] as MainController
50    component [RecommendationClientService] as RecommendationClientService
51    portout " " as out1
52    portout " " as out2
53    component [RecommendationService] as RecommendationService
54    portout " " as ClientReco
55    component [UserService] as UserService
56    portout " " as ClientUser
57    component [MovieService] as MovieService
58    portout " " as ClientMovie
59
60    ClientInterface -- ClientIn
61    ClientIn -- MainController
62    MainController ..> RecommendationClientService : requires
63    MainController ..> UserService : requires
```

```
64      RecommendationService ..ClientReco
65      UserService .. ClientUser
66      MovieService .. ClientMovie
67      RecommendationClientService .. out1
68      RecommendationClientService .. out2
69  }
70
71  ClientMovie ..> MovieInterface1 : requires
72
73  ClientUser ..> userInterface1 : requires
74  ClientReco ..> recoInterface1 : requires
75  out1 ..> userInterface1 : requires
76  out2 ..> recoInterface1 : requires
77
78  component [config-service] as config
79
80  component [eureka-service] as eureka
81
82  movieService ..> eureka : use
83  recoService ..> eureka : use
84  recoClient ..> eureka : use
85  userService ..> eureka : use
86
87  movieService ..> config : use
88  recoService ..> config : use
89  recoClient ..> config : use
90  userService ..> config : use
91  eureka ..> config : use
92
93  @enduml
```

Listing A.6: Our gold standard for the component-based architecture of the Movie Recommendation System case study. The representation of this software architecture is based on PlantUML notation.

```
1   @startuml https://github.com/spring-petclinic/spring-petclinic-microservices
2
3   interface "/owners" as ownerInterface
4   interface "/petTypes" as petInterface1
5   interface "/owners/{ownerId}/pets" as petInterface2
6   interface "/owners/*/pets/{petId}" as petInterface3
7
8   component [customer-service] as customer {
9      portIn " " as ownerIn
10     portIn " " as petIn1
11     portIn " " as petIn2
12     portIn " " as petIn3
13     component [OwnerResource] as OwnerResource
14     component [OwnerRepository] as OwnerRepository
```

```
15      component [PetResource] as PetResource
16      component [PetRepository] as PetRepository
17
18      ownerInterface -- ownerIn
19      ownerIn -- OwnerResource
20      petInterface1 -- petIn1
21      petInterface2 -- petIn2
22      petInterface3 -- petIn3
23      petIn1 -- PetResource
24      petIn2 -- PetResource
25      petIn3 -- PetResource
26      OwnerResource ..> OwnerRepository : requires
27      PetResource ..> OwnerRepository : requires
28      PetResource ..> PetRepository : requires
29  }
30
31  interface "/vets" as vetsInterface
32  component [vets-service] as vets {
33      portIn " " as vetIn
34      component [VetResource] as VetResource
35      component [VetRepository] as VetRepository
36
37      vetsInterface -- vetIn
38      vetIn -- VetResource
39      VetResource ..> VetRepository : requires
40  }
41
42  interface "owners/*/pets/{petId}/visits" as visitInterface1
43  interface "pets/visits" as visitInterface2
44  component [visits-service] as visits{
45      portin " " as visitIn1
46      portin " " as visitIn2
47      component [VisitResource] as VisitResource
48      component [VisitRepository] as VisitRepository
49
50      visitInterface1 -- visitIn1
51      visitInterface2 -- visitIn2
52      visitIn1 -- VisitResource
53      visitIn2 -- VisitResource
54      VisitResource ..> VisitRepository : requires
55  }
56
57  interface "/api/gateway" as gatewayInterface
58  component [api-gateway] as gateway{
59      portIn " " as gatewayIn
60      component [ApiGatewayController] as ApiGatewayController
61      component [CustomersServiceClient] as CustomersServiceClient
```

```
62      component [VisitsServiceClient] as VisitsServiceClient
63
64      gatewayInterface -- gatewayIn
65      gatewayIn -- ApiGatewayController
66      ApiGatewayController ..> CustomersServiceClient : requires
67      ApiGatewayController ..> VisitsServiceClient : requires
68  }
69
70  component [discovery-service] as discovery
71
72  component [config-service] as config
73
74  component [admin-server] as admin
75
76  component [Prometheus] as Prometheus
77
78  component [Grafana] as Grafana
79
80  component [zipkin] as zipkin
81
82  gateway ..> config : use
83  discovery ..> config : use
84  admin ..> config : use
85  zipkin ..> config : use
86  customer ..> config : use
87  visits ..> config : use
88  vets ..> config : use
89
90  gateway ..> discovery : use
91  admin ..> discovery : use
92  zipkin ..> discovery : use
93  customer ..> discovery : use
94  visits ..> discovery : use
95  vets ..> discovery : use
96
97  gateway ..> zipkin : Sends Tracing Data
98  customer ..> zipkin : Sends Tracing Data
99  visits ..> zipkin : Sends Tracing Data
100 vets ..> zipkin : Sends Tracing Data
101
102 @enduml
```

Listing A.7: Our gold standard for the component-based architecture of the Spring Pet Clinic case study. The representation of this software architecture is based on PlantUML notation.

```
1  @startuml https://github.com/sqshq/PiggyMetrics
2
3  component [gateway] as gateway
```

```
4
5  component [registry] as registry
6
7  component [config] as config
8
9  interface "/notifications" as notificationInterface
10 interface "/notifications/recipients" as recipientsInterface
11
12 component [notification-service] as NotiService{
13    portin " " as NotiIn1
14    portin " " as NotiIn2
15    component [RecipientController] as RecipientController
16    component [RecipientServiceImpl] as RecipientServiceImpl
17    component [RecipientRepository] as RecipientRepository
18    component [NotificationServiceImpl] as NotificationServiceImpl
19    component [AccountServiceClient] as AccountServiceClient
20    component [EmailServiceImp] as EmailServiceImp
21    component [FrequencyReaderConverter] as FrequencyReaderConverter
22    component [FrequencyWriterConverter] as FrequencyWriterConverter
23    portout " " as NotiAccount
24
25    notificationInterface -- NotiIn1
26    recipientsInterface -- NotiIn2
27    NotiIn2 -- RecipientController
28    NotiIn1 -- NotificationServiceImpl
29    RecipientController ..> RecipientServiceImpl : requires
30    RecipientServiceImpl ..> RecipientRepository : requires
31    NotificationServiceImpl ..> RecipientServiceImpl : requires
32    NotificationServiceImpl ..> EmailServiceImp : requires
33    NotificationServiceImpl ..> AccountServiceClient : requires
34    AccountServiceClient .. NotiAccount
35    RecipientRepository ..> FrequencyReaderConverter : requires
36    RecipientRepository ..> FrequencyWriterConverter : requires
37 }
38
39 interface "/accounts" as AccountInterface
40 NotiAccount ..> AccountInterface : requires
41
42 component [account-service] as AccService{
43    portin " " as AccountNoti
44    component [AccountController] as AccountController
45    component [AccountServiceImpl] as AccountServiceImpl
46    component [StatisticsServiceClient] as StatisticsServiceClient
47    component [AuthServiceClient] as AuthServiceClient
48    component [AccountRepository] as AccountRepository
49    component [StatisticsServiceClientFallback] as ⟷
          StatisticsServiceClientFallback
```

```
50     portout " " as AccountAuth
51     portout " " as AccountStatistic
52
53     AccountInterface -- AccountNoti
54     AccountNoti -- AccountController
55     AccountController ..> AccountServiceImpl : requires
56     AccountServiceImpl ..> StatisticsServiceClient : requires
57     AccountServiceImpl ..> AuthServiceClient : requires
58     AccountServiceImpl ..> AccountRepository : requires
59     AccountServiceImpl ..> StatisticsServiceClientFallback : requires
60     AuthServiceClient .. AccountAuth
61     StatisticsServiceClient .. AccountStatistic
62 }
63
64 interface "/uaa/users" as authInterface
65 interface "/statistics" as statisticsInterface
66 AccountAuth ..> authInterface : requires
67 AccountStatistic ..> statisticsInterface : requires
68
69 component [auth-service] as AuthService{
70     portin " " as AuthAccount
71     component [UserController] as UserController
72     component [UserSeriveImpl] as UserSeriveImpl
73     component [UserRepository] as UserRepository
74     component [MongoUserDetailsService] as MongoUserDetailsService
75
76     authInterface -- AuthAccount
77     AuthAccount -- UserController
78     UserController ..> UserSeriveImpl : requires
79     UserSeriveImpl ..> UserRepository : requires
80     MongoUserDetailsService ..> UserRepository: requires
81 }
82
83 interface "/latest" as StatisticsOutInterface
84
85 component [statistics-service] as StatService{
86     portin " " as StatisticAccount
87     component [StatisticsController] as StatisticsController
88     component [StatisticsServiceImpl] as StatisticsServiceImpl
89     component [DataPointRepository] as DataPointRepository
90     component [ExchangeRatesServiceImpl] as ExchangeRatesServiceImpl
91     component [ExchangeRatesClient] as ExchangeRatesClient
92     component [DataPointIdReaderConverter] as DataPointIdReaderConverter
93     component [DataPointIdWriterConverter] as DataPointIdWriterConverter
94     component [ExchangeRatesClientFallback] as ExchangeRatesClientFallback
95     portout " " as StatisticsOut
96
```

```
97     statisticsInterface -- StatisticAccount
98     StatisticAccount -- StatisticsController
99     StatisticsController ..> StatisticsServiceImpl : requires
100    StatisticsServiceImpl ..> DataPointRepository : requires
101    StatisticsServiceImpl ..> ExchangeRatesServiceImpl : requires
102    ExchangeRatesServiceImpl ..> ExchangeRatesClient : requires
103    ExchangeRatesServiceImpl ..> ExchangeRatesClientFallback : requires
104    DataPointRepository ..> DataPointIdReaderConverter : requires
105    DataPointRepository ..> DataPointIdWriterConverter : requires
106    ExchangeRatesClient .. StatisticsOut
107    StatisticsOut ..> StatisticsOutInterface : requires
108 }
109
110 gateway ..> notificationInterface : depends on
111 gateway ..> recipientsInterface : depends on
112 gateway ..> AccountInterface : depends on
113 gateway ..> authInterface : depends on
114 gateway ..> statisticsInterface : depends on
115
116 AccService ..> config : use
117 NotiAccount ..> config : use
118 StatService ..> config : use
119 AuthService ..> config : use
120 gateway ..> config : use
121 registry ..> config : use
122
123 AccService ..> registry : use
124 NotiAccount ..> registry : use
125 StatService ..> registry : use
126 AuthService ..> registry : use
127 gateway ..> registry : use
128
129 @enduml
```

Listing A.8: Our gold standard for the component-based architecture of the Piggy Metrics case study. The representation of this software architecture is based on PlantUML notation.