# Policy-Driven Authorization in Microservice-Based Applications

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

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

genehmigte

## Dissertation

von

**Niklas Ulrich Sänger**

Tag der mündlichen Prüfung: 06.02.2025

1. Referent: Prof. Dr. Sebastian Abeck
2. Referent: Prof. Dr. Kurt Geihs

# Contents

# 1 Introduction

Digitalization is an important goal for governments and companies around the world. The German government passed a law on the digitalization of public administrations, so-called E-Government, in 2013 [BJ-FEV]. This requires a large amount of new software systems, which must be interoperable with each other to provide a real benefit to end users (e.g., citizens). This will eventually result in a wide application and infrastructure landscape. While digitalization can provide great benefits for end users, such as an increased speed and efficiency in processes, security becomes an increasingly important aspect. In past years, various cybersecurity attacks on public administrations have already been carried out. For example, the German county of Anhalt-Bitterfeld suffered a ransomware attack in 2021 and took over a year to fully recover from the damages [HO22]. Meanwhile, governmental services were limited for citizens. Thus, security is an indispensable aspect when it comes to the development of software, e.g., for the public sector handling sensitive data.

At the same time as the efforts in digitalization increased, the IT landscape changed constantly. Cloud infrastructures (either private or public) with their various service models such as Infrastructure as a Service (IaaS) or Platform as a Service (PaaS) have become the standard option for companies. In 2020, a survey among German companies indicated that 82 % of the responding companies use cloud computing, which is an increase of more than 50 % from 2011 [SR22]. In parallel to the developments in cloud infrastructure, the microservice architecture became popular. Newman defines microservices as "*small, autonomous services that work together*" [Ne15]. Combined with containerization technologies, e.g., Docker, the microservice architecture fits perfectly into the cloud infrastructure [JJ+19]. Together with a container orchestration system, e.g., Kubernetes, microservice-based applications can be, among others, efficiently deployed, distributed among different regions, and scaled depending on the current load [TG20].

For the development of a microservice-based application, a structured engineering approach is required [Sc24]. This is especially necessary when aiming to create maintainable software which can be easily extended and maintained. The microservice architecture allows a system to be divided into functional (micro)services. Each microservice can be implemented in a different programming language. However, the important aspect is the definition of Application Programming Interface (API) specifications, which are a contract between different microservices. The definition of that contract can have various characteristics. Most modern API specifications are either resource-oriented (e.g., REpresentational State Transfer (REST) APIs) or function-oriented (e.g., remote procedure call using gRPC Remote Procedure Calls (gRPC)) [JS+18].

With the introduction of the microservice architecture and the deployment and operation thereof, the overall management complexity rises due to the distribution compared to a monolithic application [DG+17; WL+21]. This can result in security issues in modern web applications. The Open Web Application Security Project (OWASP) foundation released a top ten list in 2021, which lists broken access control as the top security issue in modern web applications [OW21]. Access control describes the mechanisms which control who can access a resource [SS94]. To address broken access control, access control must be considered right from the start of the development of a microservice-based application throughout all phases of the Software Development Life Cycle (SDLC) [KK+21].

This includes the integration of authentication and authorization. Authentication requires proof of a user's identity. After a user is successfully authenticated, the access to a resource can be authorized. Throughout the time, various concepts to authorize access to a resource (e.g., file) have been developed. A simple mechanism is an Access Control List (ACL), which is a list of user permissions (e.g., read, write) associated with a resource (e.g., file). In Role Based Access Control (RBAC), permissions can be assigned to roles which can be assigned to arbitrary users. This concept is further extended by Attribute Based Access Control (ABAC) which defines permissions through policies. A policy is a set of rules that must be fulfilled given the attribute values by the user (e.g., group), the environment (e.g., time), or resource (e.g., resource owner) [HF+14].

While there are sophisticated solutions to integrate authentication into a web application (e.g., using OpenID Connect (OIDC) [SB+14]), the integration of authorization into microservices is still an active research area [AC22]. One challenge is the granularity of authorization in the microservice architecture. So far, the focus has been primarily set on technical solutions towards the integration of coarse-grained authorization mechanisms, especially using OAuth2.0. To address this, Nehme et al. [NJ+18] and Sauewens et al. [SH+21] each propose an approach towards fine-grained authorization with the use of ABAC. Both approaches focus on the technical realization and create a single point of failure due to the placement of authorization components and create coupling in the microservice architecture. In addition, they do not include the systematic creation of authorization policies, which is another challenge. Brossard et al. [BG+17] propose such a general approach to the systematic implementation of ABAC authorization policies based on use case artifacts. However, the proposed approach lacks overall reproducibility and only provides a high-level overview of a systematic development. Another example is provided by Li et al. [LC+21], who propose the automatic generation of Service-to-Service (S2S) authorization policies based on a static code analysis of a microservice. However, the generated authorization policies are coarse-grained and do not consider the user context in which requests are performed.

The approaches presented in the literature lack a holistic consideration of authorization in the development of a microservice-based application through the development phases of analysis, design, and implementation and test. To fill this gap, this thesis provides the Microservice Authorization Framework (MAF) which introduces a systematic approach to policy-driven authorization into the

development of microservice-based applications. The MAF comprises three contributions and utilizes ABAC concepts to perform fine-grained authorization decisions [AQ+18]. The primary contribution is the development of authorization policies based on existing development artifacts. This is complemented by the creation of S2S authorization policies, which consider the authorization of requests occurring between microservices. The third contribution is the systematic integration of authorization components into the microservice architecture.

The remainder of this chapter is further structured as follows: Section 1.1 provides a classification of this thesis in the field of computer science. Section 1.2 introduces an exemplary scenario used for the derivation of research questions highlighted in Section 1.3. Section 1.4 describes the scientific contributions of this thesis. The premises for the scientific contributions are introduced in Section 1.5. Finally, Section 1.6 outlines the structure of this thesis.

## 1.1 Classification of This Thesis

The thesis contributes to the field of software engineering, which can be defined as "*the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software*" [ISO-247]. As presented by the Software Engineering Body of Knowledge published by the IEEE Computer Society, software engineering itself can be structured into several knowledge areas [Wa24]. This thesis comes into contact with various knowledge areas including software security, software requirements, and software architecture as outlined subsequently. The knowledge area software security aims at introducing security into the design of the software instead of patching a software afterward. Therefore, security must be considered in every stage of a development life cycle [Wa24]. An important aspect is the introduction of security into the elicitation of software requirements. Next to the collection of functional requirements determining the business logic, non-functional requirements considering security must be collected throughout the requirements analysis phase to lay the foundation for secure software. In the context of software architecture, this thesis limits its point of view to the microservice architecture style, which has become a popular architecture style in research and industry [FM+17; SL20; BG+22].

Complementing the field of software engineering is the field of cybersecurity, defined as "*the organization and collection of resources, processes, and structures used to protect cyberspace and cyberspace-enabled systems from occurrences that misalign de jure from de facto property rights*" by Stevenson et al. [CD+14]. While the field of cybersecurity is extensive, this thesis focuses on the aspects of authorization as part of access control. As defined by Sandhu and Samarati, "*access control constrains what a user can do directly, as well as what programs executing on behalf of the users are allowed to do. In this way, access control seeks to prevent activity that could lead to breach of security*" [SS94].

## 1.2 Scenario Under Consideration

This section introduces a scenario to further motivate this thesis and the related research questions. Figure 1.1 provides an overview of this scenario. The scenario considers a software company that is developing a microservice-based application for customers.

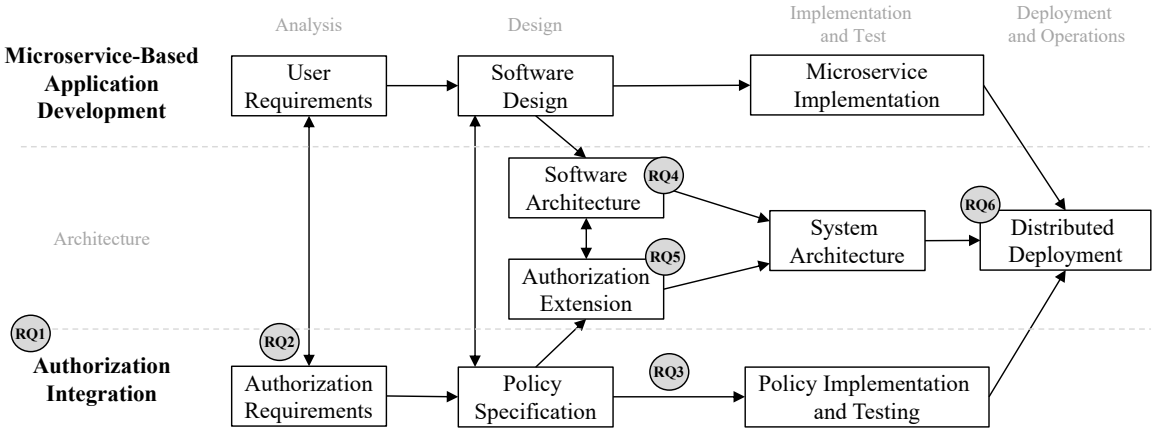


**Figure 1.1:** Scenario Under Consideration

### Initial Situation

The software company has a structured process for the systematic development of a microservice-based application, which is presented in the upper half of Figure 1.1. However, in the initial situation, the software company lacks a process for the holistic consideration of the integration of authorization in their microservice-based applications. This results in different implementations of authorization (e.g., in source code) by different development teams. Therefore, the software company is in need for a dedicated authorization integration process to streamline the authorization implementations.

The currently employed microservice development process contains the phases of analysis, design, implementation and test, and deployment and operations. In this context, the structured process creates an artifact in one phase, e.g., analysis phase, and transfers this artifact into an artifact in the subsequent phase, e.g., design phase. Furthermore, the software company follows the microservice architecture style, structuring an application into a collection of loosely coupled microservices [Ne15].

To develop a new application for a customer, the software company first establishes the user requirements together with the customer during the analysis phase. After the collection of requirements with a customer, the software company begins to realize the requirements with the design of the software during the design phase. This includes the design of the software architecture, which also takes the distribution of the microservices in the architecture into account. The use of the microservice

architecture allows the software company to design and implement microservices in parallel using several (distributed) development teams and potentially different programming languages [Ne19]. Each microservice has a web API that is known among developers. The developed microservices are finally deployed in a distributed, containerized environment such as Kubernetes. In this environment, the software company assumes an implicit trust between the microservices and therefore does not use an explicit mechanism to control access on requests occurring between microservices.

### Integration of Authorization Into Microservice-Based Application Development

To address the software company's need for a systematic integration of authorization into their development activities, the software developers should be supported throughout the phases of the development [KK+21].

To do so, an approach such as presented in the lower half of Figure 1.1 can complement the existing microservice-based development process that is already used by the software company. To also include authorization requirements as the specification of access and privileges [Fi03] during the design phase, the collection of user requirements should be extended. Depending on the existing artifacts during the analysis phase (e.g., use cases, other functional requirements), the authorization requirements can be derived from the functional requirements created together with a customer. If the existing artifacts neglect information vital for the derivation of authorization requirements, the artifacts should be adapted. For each user requirement, there should be at least one authorization requirement which defines the access.

To develop a structure preserving approach, the design phase should pick up the previously defined requirements. The user requirements can be structured into microservices which implement the business logic of the application. This structuring of the microservices results in a concrete software architecture. To enable authorization for a microservice, the authorization requirements which belong to the user requirements of a microservice should be further specified into a set of policies. Depending on the authorization mechanism (e.g., ABAC, RBAC) and the location of the authorization mechanisms, the authorization components extend the software architecture. For example, externalized authorization has implications on the software architecture as well as the deployment architecture by requiring additional components such as an API proxy [HF+14]. In addition, the implicit trust between microservices should be removed to reduce the risk of unauthorized requests by compromised microservices [CB+21].

During the implementation, the authorization policies are realized. Depending on the selected authorization mechanisms, the policy can be either implemented inside the microservice code or in a concrete policy language which can be externally evaluated (e.g., eXtensible Access Control Markup Language (XACML) or Rego). Finally, the microservice and the tools necessary to evaluate policies

should be deployed together. Therefore, existing deployment processes of the software company may be complemented to also include authorization mechanisms, realizing the concrete architecture.

## 1.3 Definition of Research Questions

This section elaborates the Research Questions (RQ) that arise from the scenario presented in the previous section. The research questions are used to set the objectives of this thesis and can be found in Figure 1.1. In total, there are six research questions marked as RQ1 to RQ6.

### RQ1 - How to systematically integrate authorization into the development of microservice-based applications?

In the considered scenario, each development team implements authorization in their microservice. Without uniform guidelines, the implementation of authorization can vary between developers. As identified in a systematic mapping study on security approaches in software engineering by Khan et al. [KK+21], only 26 % of studies considered security throughout all phases of the SDLC. To be able to develop secure software, one aspect is the support of developers by a systematic approach for the integration of authorization. This is especially relevant in a landscape in which different teams develop different microservices. Therefore, the first research question addresses the lack of a holistic integration of authorization throughout the development of a microservice-based application. To support developers throughout all phases of the SDLC and to provide a uniform approach towards authorization in an organization, this thesis provides the MAF.

### RQ2 - How to derive authorization requirements from existing analysis artifacts?

To allow a systematic definition of authorization policies, the authorization requirements for the application under development must first be defined. The definition of authorization requirements should be considered in the analysis phase of the engineering process [WA08]. A key artifact of the analysis phase are the user requirements, which are a collection of needs expressed by stakeholders using an application [SV08]. A graphical representation of user requirements is typically created using Unified Modeling Language (UML) use case diagrams [OMG-UML], which can be further specified using natural language. However, to include authorization into the development of an application, a clear understanding of which stakeholders can access a resource must be established in the form of authorization requirements with a formalized process based on existing artifacts [WA08]. Such requirements should also be defined in natural language using a predefined structure. To further support developers, this second research question seeks to elaborate on a systematic process to derive authorization requirements from existing analysis artifacts.

### RQ3 - How to implement fine-grained authorization policies?

The authorization requirements must be considered during the design phase with the goal of transforming such authorization requirements into a more detailed policy specification [WA08]. In the considered scenario, this transformation must accompany the design phase and the implementation and test phase of the microservice-based application. Existing approaches towards the implementation of fine-grained authorization into microservices are often on a technical level, specifying how to enforce the authorization decisions (e.g., [NJ+18; SH+21; BK+18]). Approaches specifying what must be authorized, i.e., the derivation of fine-grained authorization policies, do not focus on microservices (e.g., [BG+17]). This includes the missing consideration of the microservice API specification, which describes the interface a client or user can interact with [Ne15]. Thus, the third research question targets the implementation of fine-grained authorization policies from two viewpoints: First, a systematic process to derive what must be authorized in the design phase. Second, a systematic implementation of authorization policies using a suitable policy language in the implementation and test phase.

### RQ4 - How to implement S2S authorization in a microservice-based application?

Over the last few years, the concept zero trust has evolved [TU+21]. Zero trust aims to remove implicit trust from resources of an enterprise or an organization. The perimeter model is often used inside an organization and assumes that everything inside a trusted environment (e.g., network) can be trusted. This creates implicit trust. Therefore, the zero trust concept moves from a perimeter model / implicit trust to the idea of "*never trust, always verify*" [BO+21]. To address this, in 2020 the National Institute of Standards and Technology (NIST) published a zero trust architecture including a dedicated Policy Enforcement Point (PEP) and Policy Decision Point (PDP) to verify requests to resources [RB+20]. The perimeter model is also applied to microservice-based applications, e.g., deployed to a Kubernetes cluster. To remove this implicit trust from microservice-based applications, the trust between microservices must be considered when designing and implementing the application. There are existing approaches which generate authorization policies for the S2S interaction between microservices [XZ+23; LC+21]. However, these approaches result in coarse-grained authorization policies. RQ4 aims to address this coarse-granularity by providing developers a systematic implementation process for S2S authorization policies.

### RQ5 - How to externalize authorization in a microservice-based application?

Another aspect of integrating policy-driven authorization into a microservice-based application is the consideration of how the policies are enforced. In the considered scenario, each development

team implements authorization in the microservice's source code. However, with the use of ABAC, policies are enforced using components such as the PEP and PDP [OAS-XAC]. Externalizing these authorization components from the microservice allows maintaining and modifying authorization policies independent of the microservice's code. This improves a microservice's reusability and allows providing an organization wide authorization structure while implementing microservices in different programming languages. Nonetheless, existing approaches (e.g., [SH+21]) do not fully externalize the authorization logic from the microservice. Therefore, RQ5 targets the support of externalized authorization by providing an extension of the software architecture in the design phase of the development. Additionally, an approach how the necessary authorization components are deployed to a (cloud) environment is considered.

#### RQ6 - How to decentralize authorization in a microservice-based application?

The previous research question is complemented by the decentralization of the authorization mechanism in the microservice-based application. If authorization is externalized, the authorization decisions should not be performed in a centralized authorization services. This introduces coupling into the architecture and creates a single point of failure. Therefore, using a centralized authorization mechanism is considered to be a security smell [PS+21]. To counter this, RQ6 targets the decentralization of authorization. Therefore, the design and deployment and operations phases are extended to consider the decentralization of authorization components.

## 1.4 Scientific Contributions

The state of the art (see Chapter 3) presents related work to the research questions elaborated in the previous section. Based on the research questions, the scientific contributions of this thesis are established. One scientific contribution can address one or multiple research questions. The scientific contributions are collected in our MAF, which is introduced in Chapter 4. The goal of MAF is to support developers with the systematic integration of policy-driven authorization into the development of a microservice-based application. In doing so, MAF addresses RQ1 by addressing the aspect of authorization holistically. Each of the following contributions is one component of MAF and further introduced in a dedicated chapter.

### 1.4.1 C1 - Authorization Policy Development

The first contribution of this thesis is the elaboration of a development approach to derive and implement fine-grained authorization policies. The approach spans through the phases of analysis,

design, and implementation and test. The focus of this contribution is the authorization of the interaction of a user with a microservice-based application. The first step is the systematic derivation of authorization requirements, addressing RQ2. As a foundation, functional requirements defining what a user wants to do with an application are used to derive the required authorization knowledge. In the design phase, the design artifacts of a microservice are used in combination with authorization requirements to create authorization policies. The primary design artifact is the API specification, defining how to interact with a microservice. The authorization policies are defined independent of a specific policy language. In the implementation and test phase, a systematic approach towards the implementation of authorization policies is presented. The policy language Rego is used here [OP-Do], which allows, among others, the structured implementation and storage of policies. Thereby, the contribution C1 additionally addresses RQ3.

### 1.4.2 C2 - Service-to-Service Authorization Policy Development

The second contribution investigates the systematic integration of S2S authorization between microservices. This contribution complements contribution C1 by creating authorization policies for S2S requests. The S2S authorization policies are created based on authorization requirements and an in-depth analysis of the design artifacts of the microservice-based applications. An important aspect is the identification of the occurring S2S requests. In addition, C2 investigates the modifications required to the code base of a microservice to support S2S authorization. The contribution C2 primarily addresses RQ4.

### 1.4.3 C3 - Authorization Application Integration

The third contribution investigates the integration of policy-driven authorization using ABAC into a microservice-based application. As a result, the structured approach for the architectural extension and consequent deployment is developed. ABAC requires the placement of specific components in the architecture. To enforce policies, the Policy Enforcement Point (PEP) must be placed before a request reaches a microservice. Thus, to perform the authorization externalized, the PEP component must be placed in front of a microservice, addressing RQ5. The PDP must decide if a request to a microservice is valid. To guarantee that a request can be evaluated as fast as possible to assure quick response times, the PDP must be located close to the PEP. Therefore, C3 provides an approach to decentralize the authorization mechanism close to a microservice and to distribute authorization policies, addressing RQ6.

### 1.4.4 Demonstration of the Contributions

The demonstration of the contributions provided by MAF is based on the application of the contributions to an exemplary microservice-based application called CarRentalApp. Throughout this thesis, the microservice-based application CarRentalApp will be used as an example when introducing the respective contributions.

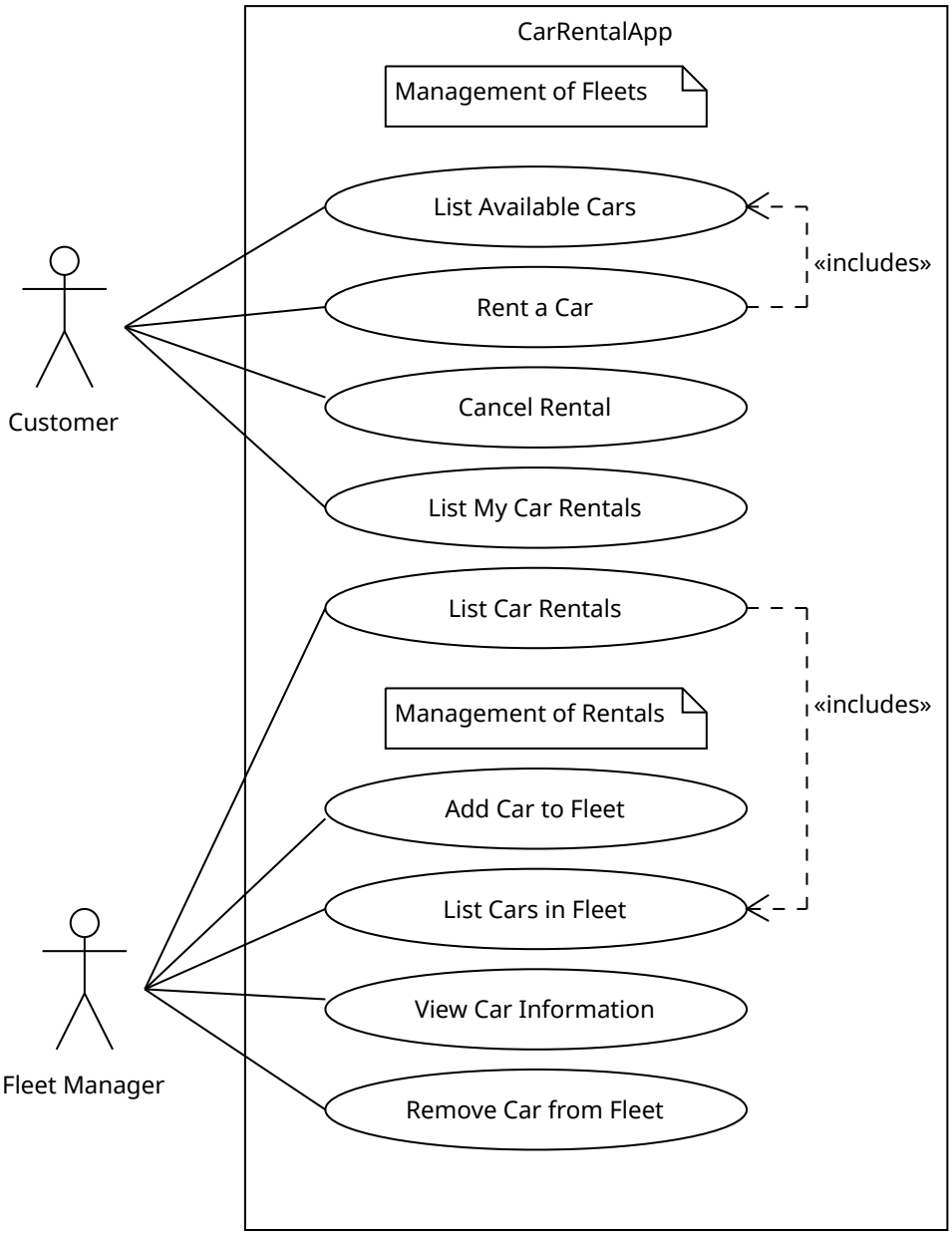


**Figure 1.2:** Functional Scope of the CarRentalApp

The CarRentalApp is an application used by a fictitious company to rent cars to customers and manage a set of cars in a fleet. To structure the functional requirements of the CarRentalApp, use cases are utilized. Figure 1.2 presents the UML use case diagram, which depicts an overview of the functionality provided by the CarRentalApp. The application has two actors, a customer and fleet manager. The customer wants to rent a car. Therefore, they first have to list available cars and rent a selected car. In addition, customers can list their rentals and cancel an upcoming rental. The fleet manager is responsible for managing the fleet of cars. Thus, the fleet manager wants to add cars to a fleet, list the cars in a fleet, view information about a car in a fleet or remove a car from a fleet. In addition, the fleet manager wants to list the upcoming and past rentals of a car in their fleet.

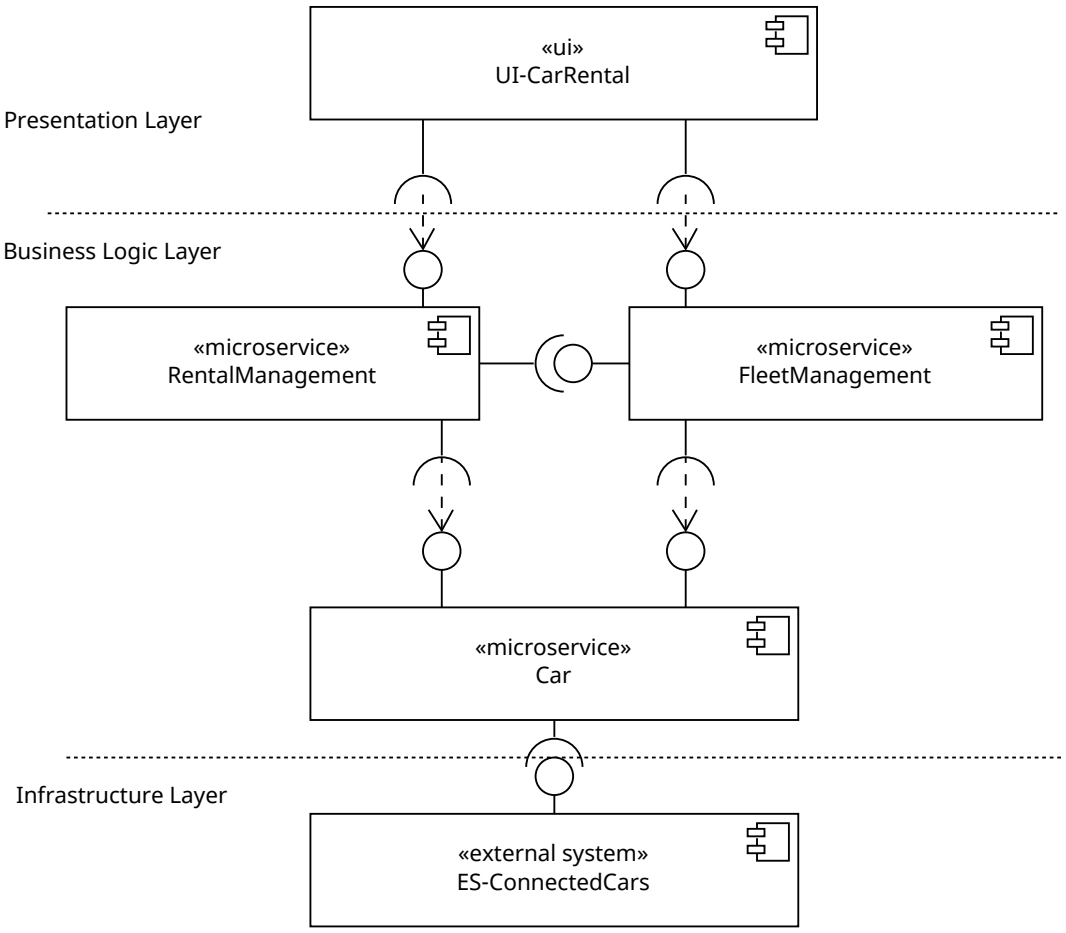


**Figure 1.3:** Component Diagram of the CarRentalApp

The use case diagram depicted in Figure 1.2 is realized using a systematic microservice-based application engineering approach proposed by Schneider [Sc24]. The use cases are structured into the capabilities *Management of Rentals* and *Management of Fleets* which are each realized by microservices. These microservices contain the business logic of the microservice-based application.

Figure 1.3 presents the software architecture as an UML component diagram of the CarRentalApp. For the modeling, the UML profile introduced by Schneider [Sc24] is used. In the presentation layer, the User Interface (UI) UI-CarRental is located. The business logic layer is comprised by the microservices RentalManagement and FleetManagement. In addition, the microservice Car provides information about a Car (e.g., brand, model), which is retrieved through a third-party service ES-ConnectedCars provided by a car manufacturer. Each microservice provides an interface through an API which either follows a resource-oriented approach (e.g., REST) or a function-oriented approach (e.g., gRPC). As presented in Figure 1.3, the microservices are either accessed by a user through the UI or by another microservice (i.e., S2S requests). For example, the microservices RentalManagement and FleetManagement both access the microservice Car to retrieve vehicle data.

## 1.5 Premises

This section describes the premises which are set for this work. The premises narrow the problem domain of this thesis.

### 1.5.1 Premise 1 - Attribute Based Access Control

There are several mechanisms to perform authorization, such as RBAC or ABAC. In order to further focus this thesis, the policy-driven authorization is done exclusively with ABAC as an emerging access control mechanism [SO17; HF+14]. ABAC allows realizing fine-grained authorization decisions [AQ+18] with the help of authorization policies and has been proposed for the use in microservices by Yarygina and Bagge [YB18]. In addition, the authorization is to be executed externalized to reduce the coupling between the implementation of the authorization and the application logic. The documentation of the authorization logic is to be carried out in the form of ABAC authorization policies.

### 1.5.2 Premise 2 - Microservice-Based Application

There are various architecture styles that can be used to develop an application. For a long time, applications were developed as monoliths. Now, practitioners vastly migrated to developing software using the microservice architecture style [SL20]. Microservice-based applications are of distributed nature and consist of multiple self-contained services. This thesis focuses on the systematic integration of authorization into the engineering of microservice-based applications.

### 1.5.3 Premise 3 - Structure-Preserving Software Engineering Process

To integrate authorization into the engineering of microservice-based applications, an existing engineering approach is required as a foundation (e.g., [Sc24]). To ensure the maintainability of the developed software, the engineering process should be structure-preserving. This provides traceability between software artifacts of different development phases [SV08]. In the context of requirements, Gotel and Finkelstein define traceability as "*the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins. through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)*" [GF94, p. 4].

### 1.5.4 Premise 4 - Definition of Structured Functional Requirements

In the analysis, there are several ways to capture user requirements. One often used artifact are use cases, which display the interaction between an actor and a system to achieve a goal [Co00]. Use cases can be, among others, further extended to capture pre- and post-conditions. To integrate authorization into microservice-based applications, functional requirements such as use cases are required as a key analysis artifact to understand what must be authorized in the microservice-based applications.

### 1.5.5 Premise 5 - Modeling of Software Artifacts Using UML

To model software development artifacts, this thesis uses UML. UML provides a broad spectrum of static and dynamic diagrams [OMG-UML]. Furthermore, the default UML specification can be extended using additional metamodels and profiles [FV04]. This also allows generating further artifacts (e.g., code stubs) based on the UML model. In this thesis, UML is used on a semantic level to convey content to the reader.

### 1.5.6 Premise 6 - Manual Creation of Development Artifacts

There are various approaches to automate the generation of authorization artifacts (see Chapter 3). This includes, but is not limited to, the use of natural language processing to extract authorization relevant aspects from natural language documents or model-driven approaches to generate authorization related artifacts (e.g., [XP+12]. However, these approaches are limited by the granularity or the completeness of the policies. In this thesis, the creation of development artifacts is considered to be a manual task performed by a software developer. In the context of policy-driven authorization, the goal is to understand which steps must be executed to create fine-grained authorization policies. For instance,

establishing the knowledge of what must be authorized. Automating the manual processes established in this thesis is considered as future work.

## 1.6 Structure of this Thesis

Figure 1.4 depicts the structure of this thesis. Chapter 2 provides the necessary foundations for this thesis. This includes concepts of software engineering as well as access control. The state of the art is presented and assessed in Chapter 3. Based on the assessment of the state of the art using a requirements catalog, the research gaps are elaborated and addressed in the subsequent chapters.

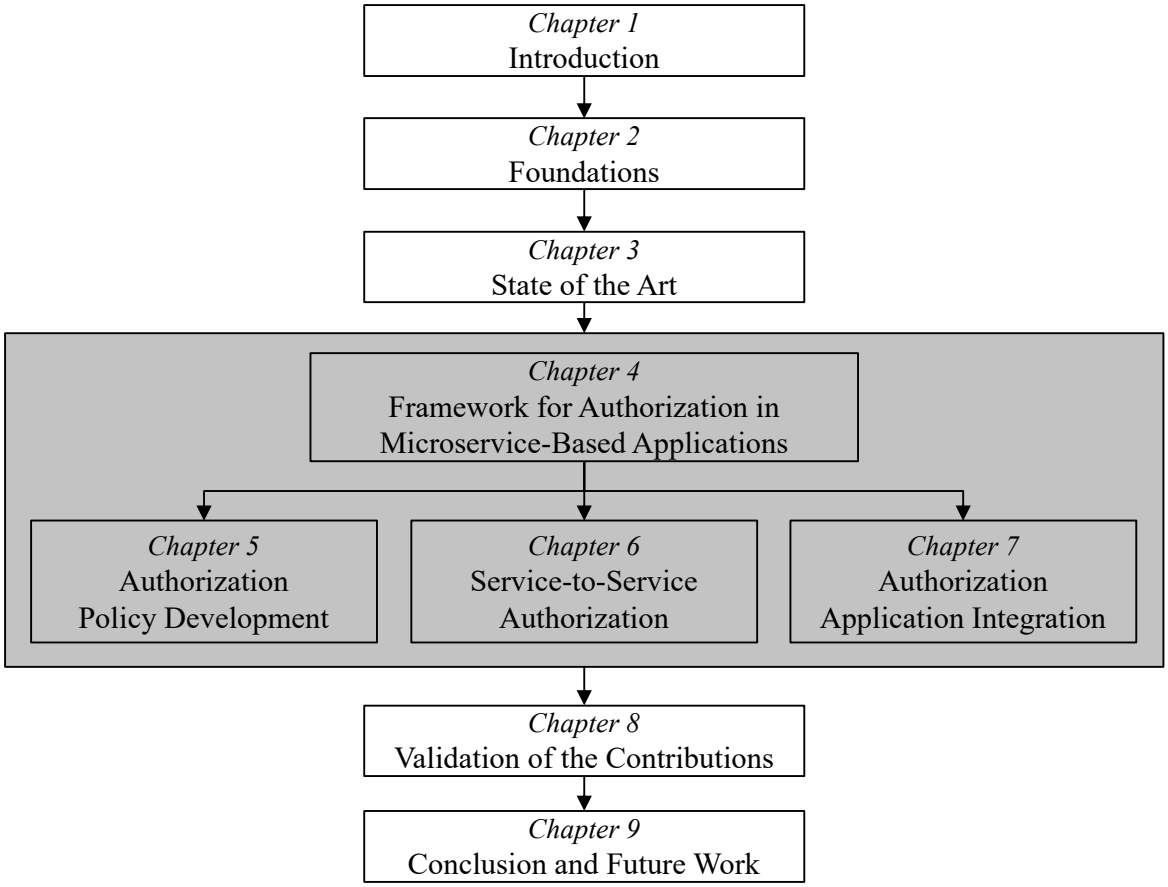


**Figure 1.4:** Structure of the Dissertation

Chapters 4 to 7 present the primary contributions of this thesis. The overarching contribution is the MAF presented in Chapter 4. MAF structures the subsequent contributions and provides the context in which they operate. All contributions of MAF follow the phases of the SDLC, namely, analysis, design, implementation and test, and deployment and operations. Thus, each contribution presented in Chapters 5 to 7 is in itself structured along the respective development phases. Chapter 5 introduces

the development of authorization policies. The basis for the development of the policies is provided by existing software development artifacts of a microservice-based application. These authorization policies are complemented by S2S authorization policies, which are introduced in Chapter 6. Finally, to realize the created authorization policies, the microservice-based application must be adapted. This includes, among others, the modification of the overall architecture, which is introduced in Chapter 7.

The contributions of MAF are validated in Chapter 8 using a case study on the example microservice-based application CarRentalApp. Finally, a conclusion of this thesis and future research directions are provided in Chapter 9.

# 2 Foundations

This chapter introduces the foundations of this thesis. As presented in Section 1.1, this thesis is placed in the context of software engineering and cybersecurity. Therefore, the first part of this chapter introduces software engineering concepts in Section 2.1 and more specifically the microservice architecture in Section 2.2. The aspects of cybersecurity are addressed in the subsequent sections, including an introduction of authentication in Section 2.3, access control in Section 2.4, and a detailed introduction of Attribute Based Access Control (ABAC) in Section 2.5. Finally, Section 2.6 introduces the concept of zero trust.

## 2.1 Software Engineering

Software engineering is a broad research field containing various sub categories [Wa24]. This section provides a brief overview of the core concepts required for this thesis. This includes the introduction of software development phases and an overview of a Software Development Life Cycle (SDLC).

The need for a systematic and structured development of software has existed for a long time. In 1968 and 1969, NATO held conferences on software engineering to develop guidelines and best practices that can still be found today [PB68]. A fundamental aspect of software development is the structuring of development into different phases. These phases include the (requirements) analysis, design, implementation and test, and deployment and operations [Vi07]. These phases can have different names and be ordered differently, depending on the selected SDLC model [Ru10]. One of the first models is the waterfall model, also known as the cascade model. The lifecycle phases are executed one after another, with or without a feedback loop [Ro70]. More modern approaches introduce agility into the development, e.g., by having short development cycles implementing only selected features [AS+02].

In the analysis phase, the functional and non-functional requirements of the application are collected. This defines what should be developed and is therefore important to "*build the right thing*" [Sm15, p. 8]. In the design phase, the overall software is designed by taking the requirements into account. The implementation and test phase realizes the created design. Additionally, the implementation is tested to create a high software quality. Finally, the software is deployed to a production environment. In the context of authorization, especially the analysis and design phase are of interest as they dictate the

creation of authorization artifacts. Hence, the following subsections introduce these phases in more detail.

### 2.1.1 Analysis

In the analysis phase, the requirements for an application are collected. Requirements engineering in itself is considered a knowledge area in software engineering [Wa24]. Creating and documenting requirements involves various stakeholders and processes [PR15]. This thesis will focus on functional requirements and non-functional requirements. Functional requirements "*specify observable behaviors that the software is to provide - policies to be enforced and processes to be carried out*" [Wa24, p. 46]. Non-functional requirements "*in some way constrain the technologies to be used in the implementation*" [Wa24, p. 46]. The specification of quality requirements, such as reliability, usability, or availability, are specified by non-functional requirements [Gl07]. This also includes the specification of security as a quality aspect of software.

**Use Cases** Use cases are an option to structure and specify functional requirements. According to Cockburn, a use case "*[. . . ] is a description of the possible sequences of interactions between the system discussed and its external actors, related to a particular goal*" [Co00, p. 15]. In the context of a use case, the system can be considered a black box. Cockburn identifies several types of use cases, such as blue-level or user-goal level use cases. These user-goal level use cases are used by the CarRentalApp.

Use cases are described using a textual use case description. A template structuring the use case description can be used to create a consistent description. An important aspect is the definition of actors, which can be differentiated into primary and secondary actors. In user-goal level use cases, primary actors want to achieve a certain goal. The primary actors initiate the use case. Secondary actors might be involved to achieve this user goal.

The user goal can be restricted by including conditions that must be fulfilled. This includes conditions that must be fulfilled before the use case is executed, as well as conditions that should be met after the execution of the use case. The main aspect of a use case description is the specification of flows. A flow introduces the steps that must be performed by the primary / secondary actors or the system to successfully achieve the user goal. In addition, alternative flows can be defined, to handle errors occurring during the execution of the use case. The use case description can be adapted depending on the employed environment. These use case descriptions are also known as fully dressed use cases [Co00].

As presented in Figure 1.2, use cases can also be modeled using Unified Modeling Language (UML). Use case diagrams are static UML diagrams that include the system that is to be modeled, primary

and secondary actors, and use cases [SS+15]. They allow modeling use cases on a higher abstraction level. In addition, use cases diagrams allow use cases to be linked to each other using includes or extends relationships [OMG-UML].

### 2.1.2 Design

Similar to the analysis phase, the design of a software includes a broad spectrum of activities. The design phase "*[...] is the application of software engineering discipline in which software requirements are analyzed to define the software's external characteristics and internal structure as the basis for the software's construction*" [Wa24, p. 83]. This includes the design of the software architecture. During the design phase, the decision of how the software architecture should be structured to best support the previously created functional and non-functional requirements is made. A popular architecture that can be selected is the microservice architecture, which is further introduced in the next section. In addition, the design phase includes an external-facing design of the system and its components and an internal-facing design [Wa24].

To support the design process, design patterns reflect best practices used to solve a problem. These patterns can be categorized into creation patterns, structural patterns, or behavioral patterns. In the context of microservices, a popular collection of design patterns is provided through Domain-Driven Design (DDD) by Erik Evans [Ev03]. DDD allows structuring bounded contexts into microservices [HG+17; Sc24].

An important aspect of the design phase is the modeling of design artifacts, such as component diagrams or sequence diagrams. Again, UML provides a broad spectrum of static and dynamic diagrams, that are commonly used for this purpose [OMG-UML].

## 2.2 Microservice Architecture

Following the Service-oriented Architecture (SOA), the microservice architecture is the latest distributed architecture style, which has evolved over the last decade. According to Zimmerman [Zi17], the microservice architecture is a form of SOA. The term microservice appeared in the early 2010s and was influenced by Lewis and Fowler [FL14].

> *In short, 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, often 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 and Lewis, [FL14])

The microservice architecture consists of a number of different microservices. Each microservice provides a set of functionality through a well-defined Application Programming Interface (API). Various API types and paradigms can be used by the microservices (see Section 2.2.1). In the context of a microservice-based application, the application is comprised by multiple microservices. In that case, each microservice provides a subset of the business logic of the overall application. When designing microservices, the principles of loose coupling and high cohesion must be considered, to remove dependencies between microservices [Ne15]. There is not a clear definition of how much functionality should be provided by a microservice to still consider it a microservice. However, since microservices are used in modern cloud environments [JJ+19], they should be lightweight to allow for fast horizontal scaling. Therefore, when developing microservices, cloud native principles should be considered (see Section 2.2.2).

Another potential advantage of developing microservices is organizational. The different microservices of a microservice-based application can be implemented independent of another [Ne15]. This allows to distribute the responsibility of the development to different teams [DG+17]. However, this requires that the structure of the microservice-based application has been defined, including the API specification required for the microservices to communicate. The API will hide the underlying implementation of a microservice. Hence, different development teams can use programming languages best suited for the respective task.

### 2.2.1 Microservice API

The most important feature of a microservice is the API it provides. These APIs are commonly referred to as web APIs, as they use HTTP as an underlying protocol. The specification of APIs is important when designing a microservice because it defines how the microservice can be used. Several works considered the systematic design of microservice APIs by providing guidelines and best practices [Gi18; Sc24]. In the context of this thesis, resource-oriented APIs and function-oriented APIs are further presented.

**REST** REpresentational State Transfer (REST) is a paradigm for the communication of a client with a server that has been proposed by Roy Fielding [Fi00]. Fielding defines multiple concepts for REST. The most important are: First, addressable resources. A resource is a set of information that can be addressed by a Uniform Resource Locator (URL). Second, a unified restricted service interface, which provides a fixed number of methods that can be applied to the resources. Third, stateless communication. Each request from a client to a server must contain all information required

for the server to provide a response. Fourth, representation oriented. A representation can depend on the respective client. This allows the server to respond, e.g., using a JavaScript Object Notation (JSON) or plain HTML. Finally, format-driven state transfer also known as the hypermedia concept, which requires a client to only know the first URL to the server. This is used to control the state transitions.

REST is typically used with HTTP as the underlying protocol. Web APIs following all REST concepts are also known as RESTful APIs. However, the hypermedia concept is often neglected when specifying a RESTful API. To specify a RESTful API, the OpenAPI specification language is the de facto standard [Ope-Spe]. The API can be specified using JSON or a YAML Ain't Markup Language (YAML) file. Each OpenAPI specification consists of three parts: First, *info*, containing metadata. Second, *path*, describing the HTTP paths to the respective resources. Third, *definitions*, defining reusable parts of an API, such as request or response types. OpenAPI specification are commonly visualized using a tool called SwaggerUI [Ope-Spe].

**RPC** Another API paradigm are Remote Procedure Calls (RPC). RPCs are not a new concept. In 1984, Birrell and Nelson describe RPCs as a mechanism "*[. . . ] to provide for transfer of control and data across a communication network*" [BN84, p. 1]. If an RPC is invoked, the procedure is executed at the called environment. The results are returned to the client performing the RPC.

This concept is also present in the microservice architecture [Ne15]. A popular framework to embed RPCs in a microservice is gRPC, which has been originally developed by Google [GA-Do]. gRPC uses HTTP/2 as the underlying protocol. Besides performing unary RPCs, gRPC allows performing client-side, server-side, or bidirectional streaming. To specify a gRPC API, protocol buffers are used [Go-Pro]. Protocol buffers, also known as protobuf, are a language and platform neutral mechanism for the serialization of data. A gRPC specification using protobuf consists of one or multiple services, each service containing one or multiple RPCs. Each RPC requires an input message and an output messages. Messages are structured types containing one or multiple fields. The protobuf specifications are compiled into client and server stubs for a targeted programming language (e.g., Java, Golang).

### 2.2.2 Cloud Native

Deploying a (microservice-based) application to a cloud environment has become a popular option [SP17]. To deploy the microservices, technologies such as Docker are used to package a microservice into a container that can be quickly started and stopped. The term cloud native has evolved from the introduction of cloud infrastructure. It includes cloud native principles (e.g., operation on automation platforms), architectures (mainly service-based architectures), properties (e.g., horizontal scalability, elasticity), and accompanying methods and patterns (e.g., DevOps) [KQ17].

The twelve-factor app by Wiggins introduces factors for a successful development of Software as a Service (SaaS) applications [Wi12]. These factors also apply towards the development of (cloud native) microservices, such as the factors config (store configuration in environment variables), backing services (treat backing services as attached resources), and processes (execute apps as stateless processes). By applying these factors, the microservices can be deployed to a cloud environment orchestrated by a platform such as Kubernetes [Ku-Doc].

## 2.3 Authentication

Authentication tries to establish the identity of a (human or non-human) subject [SS94]. After a successful authentication, two subjects believe that they are communicating with each other and not with a hostile third-party [BA+90]. A simple example is the identification of a person with an identity (ID) card. After the verification of the ID card (e.g., by comparing the picture), a person can be considered as authenticated. In this case, a trust exists that the ID card (e.g., issued by a government) is valid. In the context of IT systems, a user typically provides one or multiple passwords or other biometric features (i.e., fingerprint) to authenticate themselves to a system.

Throughout the time, various protocols and mechanisms have been developed to support the process of authentication in IT systems. For instance, the Lightweight Directory Access Protocol (LDAP) or Kerberos [NY+05; Se06]. These protocols are commonly used within an organization. With the emergence of web applications, additional protocols formed to support a browser-based Single Sign-On (SSO). In the early 2000s, the Security Assertion Markup Language (SAML) was released by the Organization for the Advancement of Structured Information Standards (OASIS) [HM05]. SAML is an XML-based markup language. It defines three roles: the subject (i.e., human or non-human user), the identity provider, and the service provider. Fundamentally, the subject requests the access to the service provider, which in turn requests a authentication assertion from the identity provider. The identity provider can specify various authentication methods such as username and password or even use directory services such as LDAP.

In 2014, the authentication protocol OpenID Connect (OIDC) was published by the OpenID Foundation [SB+14]. OIDC is an authentication protocol based on the OAuth 2.0 protocol [Ha12] and allows developers to perform a range of log-in flows (provided by OAuth2.0). Similar to SAML, three parties are involved to perform authentication: The end user (i.e., subject or principal - these terms are often used synonymously), the Relying Party (RP), and the OpenID Provider (OP). Listing 2.1 presents the abstract flow. First, the relying party (e.g., a single page application) initiates an authentication request to the OP. The end user is typically redirected to a login page provided by the OpenID provider to authenticate themselves, e.g., using a username and password. After the successful authentication, the user is redirected to the RP including an ID token and an access token. Using the ID token, the

RP can request user information from the UserInfo endpoint of the OpenID provider in "*a REST-like manner*" [SB+14].

```
+--------+                                  +--------+
|        |                                  |        |
|        |---------(1) AuthN Request-------->|        |
|        |                                  |        |
|        |  +--------+                      |        |
|        |  |        |                      |        |
|        |  |  End-  |<--(2) AuthN & AuthZ-->|        |
|        |  |  User  |                      |        |
|   RP   |  |        |                      |   OP   |
|        |  +--------+                      |        |
|        |                                  |        |
|        |<--------(3) AuthN Response--------|        |
|        |                                  |        |
|        |---------(4) UserInfo Request----->|        |
|        |                                  |        |
|        |<--------(5) UserInfo Response-----|        |
|        |                                  |        |
+--------+                                  +--------+
```

**Listing 2.1:** OpenID Connect Flow [SB+14]

The tokens are typically formatted using the JSON Web Token (JWT) format. Furthermore, the tokens are signed by the OpenID provider, which allows the relying party to verify the signature of the token, thereby confirming its validity. The tokens contain claims that are provided by the OpenID provider. The OIDC specification contains a set of standard claims such as subject identifier (sub), birthdate, or name. Moreover, custom claims can be specified by the OpenID provider and included into the respective tokens. In this thesis, OIDC is used as an example protocol to authenticate users of the CarRentalApp. Therefore, the JWT and more specifically the claims (i.e., attributes) provided by the authentication will be used for authorization (see Section 5.3) by our Microservice Authorization Framework (MAF).

## 2.4 Access Control

Access control is an elementary aspect of security in IT systems [Go10]. The goal of access control is to limit the access of a subject to a specific system or resource. As before, a subject can be a human user or another system. As depicted in Figure 2.1, access control is accompanied by other security aspects. The includes authentication, a reference monitor, and an authorization database. In addition, auditing is necessary to monitor the access requests [SS94].

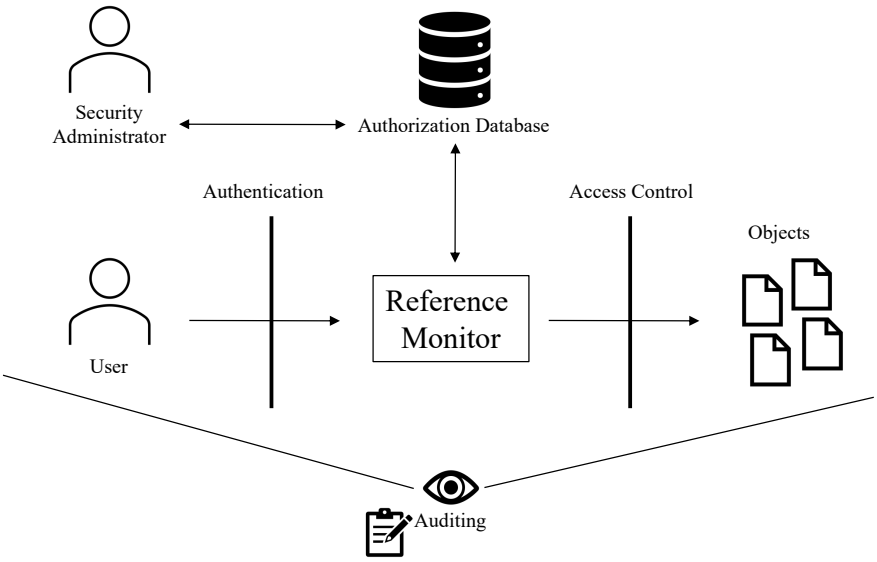


**Figure 2.1:** Overview of Access Control and Complementing Services [SS94]

Access control assumes that the authentication of a subject (i.e., user) has been successfully performed [SS94]. Authorizations specify what a subject can do in a system [Gu02]. In 1972, James Anderson introduced the concept of a references monitor in the context of operating systems, which enforces authorized access relationships between subjects and objects [An72]. The reference monitor can be considered a component in an operating system which mediates all accesses to objects or resources. Access requests can have various forms. In the context of operating systems, access requests can be read, write, or execute. The reference monitor enforces authorizations defined, e.g., in an authorization database or authorization policies.

Authorization can be differentiated based on the level of granularity, which "*[. . . ] means the level of details an authoring process requires to limit and separate privileges*" [Ki15, p. 17]. Authorization is often divided into fine-grained authorization and coarse-grained authorization [Ki15]. Coarse-grained authorization focuses on the basic ability to interact with the system resources (i.e., objects). In contrast, fine-grained authorization allows flexible access rights to specific resources for individual users [GP+06; GF13]. This thesis utilizes ABAC as an access control model, as it provides fine-granular authorization and flexibility [AQ+18].

The final aspect of access control is auditing. Auditing allows analyzing accesses performed by a user a posteriori. This enables system administrators to detect flaws in the security system or detect misuses by a user.

### 2.4.1 Access Control Models

There are different implementations and models for access control. These models can vary between the overall complexity and the granularity of authorization. The following paragraphs present the models Access Control List (ACL) and Role Based Access Control (RBAC) as an predecessor to ABAC.

**Access Control Lists** ACLs are a simple form to implement access control [SF+02]. For every object, e.g., a file, an explicit list of subjects and the respective permissions of a subject are created. This allows to create very detailed authorizations. However, managing very fine-grained ACLs for numerous users can become very complex. This can be countered by assigning users to a group which receives an ACL. Another option is the assignment of permissions to roles, which leads to RBAC described in the next paragraph.

**Role Based Access Control** In RBAC, a "*role is a semantic construct forming the basis of access control policy*" [SC+96, p. 1]. Roles can, e.g., be created by a system administrator and relate to a job function performed in a company. These roles can be connected to a set of permissions and assigned to a human user. Sandhu et al. [SC+96] define a base model called $RBAC_0$, which is presented in the following list.

- Users $U$
- Roles $R$
- Permissions $P$
- Sessions $S$
- Permission Assignment $PA \subseteq P \times R$, many-to-many permission-to-role assignment relation
- User Assignment $UA \subseteq U \times R$, many-to-many user-to-role assignment relation
- $user$: $S \rightarrow U$, function mapping each session $s_i$ to the single $user(s_i)$
- $roles$: $S \rightarrow 2^R$, function mapping each session $s_i$ to a set of roles $roles(s_i)\{r \mid (users(s_i), r) \in UA$ and session $s_i$ has permissions $\cup_r \in roles(s_i)\{p \mid (p,r) \in PA\}$

Sandhu et al. consider a user a human being [SC+96]. A role is a job function within an organization. The permissions allow a holder to perform an action in a system. A permission can be assigned to one or multiple roles. Additionally, a user can be assigned one or multiple roles. In a session, the user is mapped to the roles, which allows retrieving the respective permissions for that session. A user can

also have multiple sessions simultaneously. RBAC can also be extended. For instance, Sandhu et al. present an extension with a role hierarchy ($RBAC_1$), constraints ($RBAC_2$), or both ($RBAC_3$).

Analogous to the amount and size of ACLs required for fine-granular authorization, in RBAC the amount of overall required roles increases with the degree of granularity. If the amount of roles increases, managing the roles becomes more complex. This is a limitation of RBAC, also known as role explosion [EK10]. For instance, if RBAC should be used for fine-grained authorization, a role must be created for every permutation of a user, an object, and a permission. This can already be a hurdle for small applications with a small user base. While there are other RBAC models to combat these problems (e.g., by including a role hierarchy), this problem is inherent to RBAC [EK10]. Thus, to introduce fine-granular authorization, ABAC introduced in the next section can be used.

## 2.5 Attribute Based Access Control

ABAC is selected as the access control mechanism in this thesis. ABAC allows performing complex authorization decisions based on a set of attributes. Compared to other access control models, ABAC can be seen as a more generalized concept of RBAC (i.e., attribute role) [HF+14]. There are multiple definitions of ABAC. For this thesis, the well-established National Institute of Standards and Technology (NIST) publication *Guide to Attribute Based Access Control (ABAC) Definition and Considerations* by Hu et al. [HF+14] is selected as a foundation. The NIST standard defines ABAC as the following:

> *"**Attribute Based Access Control**: An access control method where subject requests to perform operations on objects are granted or denied based on assigned attributes of the subject, assigned attributes of the object, environment conditions, and a set of policies that are specified in terms of those attributes and conditions."* (Hu et al. [HF+14, p. 17])

The subject is the entity that aims to perform an operation on an object. A subject can be a human user or a non-human user, e.g., a service. The term user is often used synonymously with the term subject (e.g., [SO17]) or principal. An object is referred to be a system resource [HF+14]. The access to the object is managed by ABAC. An object can be anything on which an operation can be performed. Since microservice-based applications are considered in this thesis, these are generally entities of the application (e.g., provided through an API).

While the NIST publication refers to operations that are performed on an object, this thesis uses the term action to describe operations, which is also used by other publications (e.g., [CW+18; AT+19]). These actions include (but are not limited to) Create Read Update Delete (CRUD) operations on an object. The attributes are a property of the subject, the object, or the environment. The environment represents the context in which a request occurs. This might include attributes such as the time or

the location. Attributes can be considered as a set of key value pairs (e.g., *location=Karlsruhe*). Following the NIST specification, actions do not have attributes.

The NIST definition of a policy, which is employed in this thesis, is presented in the following quote:

> *"**Policy**: Policy is a representation of rules or relationships which allows determining the result of a request based on the values of attributes of subject, objects and possibly environment conditions."* (Hu et al., [HF+14, p. 17])

Thereby, a policy represents a set of enforceable rules. In turn, a rule defines complex conditions using attributes. Hu et al. further specify policies as Natural Language Policy (NLP) or a Digital Policy (DP) [HF+14]. NLPs are human expressions that can be translated to machine-enforceable access control policies. DPs are access control rules that compile into machine executable codes or signals. Subject/object attributes, actions, and environment conditions are fundamental elements of DP enforced by an access control mechanism. However, Hu et al. [HF+14] do not specify a formal model of ABAC. An overview of the formal model used in this thesis is provided in the next section.

### 2.5.1 ABAC Model

The NIST definition presented by Hu et al. only provides a high-level overview of ABAC and does not provide a formalized model. As identified by Servos and Osborn [SO17], there is not a single uniformly accepted ABAC model. Instead, there are various ABAC models that formalize the elements required by ABAC in a slightly different way. In a state-of-the-art review, Servos and Osborn list various ABAC models including general models, domain-specific models (e.g., for web services [YT05]), and hybrid models (e.g., role-centered).

As identified by Servos and Osborn, the proposed model by Yuan and Tong is the most notable in the domain of web-services [SO17]. Therefore, the model will be used as a basis for this thesis. However, the model by Yuan and Tong is adapted in two cases for this thesis. First, the model by Yuan and Tong does not use objects (as used by NIST). Instead, resources, such as web services, are accessed by subjects. For this thesis, the term object is used instead of resources to align with the NIST definition presented in the previous section. Second, the model does not consider *action* as part of the policy. Instead, Yuan and Tong only consider the action *can_access* [YT05]. The introduction of an action, based on the model of Yuan and Tong, has been established by Shu et al. [SS+09]. The model presented in the following is based on Yuan and Tong's and Shu et al.'s model but includes an action to differentiate between different requests. To comply with the NIST definition, actions have no attributes.

- Subjects $S = \{Alice, Bob, ...\}$

- Actions $A = \{create, read, delete, modify, ...\}$
- Objects $O = \{Rental, Car, ...\}$
- Environments $E$
- Subject Attributes $SA = \{age, gender, ..., SA_k\}(1 \leq k \leq K)$
- Object Attributes $OA = \{owner, color, ..., OA_m\}(1 \leq m \leq M)$
- Environment Attributes $EA = \{time, location, ..., EA_n\}(1 \leq n \leq N)$
- Attribute Assignments relations for subject s, resource r, and environment e:
  $ATTR(s) \subseteq SA_1 \times SA_2 \times SA_K$
  $ATTR(o) \subseteq OA_1 \times OA_2 \times OA_M$
  $ATTR(e) \subseteq EA_1 \times EA_2 \times EA_N$
- Policy for subject $s$, action $a$, object $o$, and environment $e$
  $P_i(s,a,o,e) = f(ATTR(s), a, ATTR(o), ATTR(e))$

Each subject in $S$ has various attributes that are assigned through an attribute assignment relation $ATTR(s)$. To access a subject attribute, Yuan and Tong use a function notation. For instance, $age(Alice) = 42$ can be used to access the attribute *age* for subject *Alice*. The object and environment attributes can be accessed analogous. An authorization policy is a boolean function for a subject $s$, action $a$, object $o$, and environment $e$. The policy depicted in Equation (2.1) presents an example of a subject that must have the role *customer* to perform the action *List* on an object *Cars* under the conditions, that the owner of the Cars is equal to the subject's name.

$$\begin{aligned} Policy(s,a,o,e) \leftarrow (Role(s) == \text{"}Customer\text{"}) \wedge (a == \text{"}List\text{"}) \\ \wedge (o == \text{"}Cars\text{"}) \wedge (Owner(o) == Name(s)) \end{aligned} \tag{2.1}$$

ABAC provides a spectrum of models ranging from general ABAC models [SO17], domain-specific models to hybrid models [VJ22; SO17]. However, the core of ABAC is the use of attributes belonging to a subject, object, or the environment to perform an authorization decision. The following paragraph will provide a brief overview of further ABAC models.

**Further ABAC Models** In 2004, Wang et al. introduced a logic-based from for ABAC [WW+04]. However, the model does not consider object or environmental attributes and focuses on the modeling of policies and their evaluation. Jin et al. introduced a role-centric ABAC model in 2012 [JS+12]. This formal model aims at restricting permissions of a role via attributes. Another model is proposed by Servos and Osborn in 2015, introducing Hierarchical Group and Atrribute-Based Access Control (HGABAC) [SO15]. HGABAC introduces a hierarchical representation of object and subject attributes.

eXtensible Access Control Markup Language (XACML) is often cited as an ABAC model [SO17]. However, XACML is only a policy language supporting fine-granular authorization using attributes and lacks a formal ABAC model [SO17]. The policy language is developed by OASIS [OAS-XAC] and the latest version is V3.0 released in 2013. XACML includes a reference architecture that is introduced in the next Section 2.5.2. The policy language is further addressed in Section 2.5.3.

### 2.5.2 XACML Reference Architecture

XACML introduces a reference architecture, also known as data flow-model, which is also referred to by the NIST special publication on ABAC by Hu et al. [HF+14]. Figure 2.2 depicts a simplified version of the reference architecture introduced by XACML.

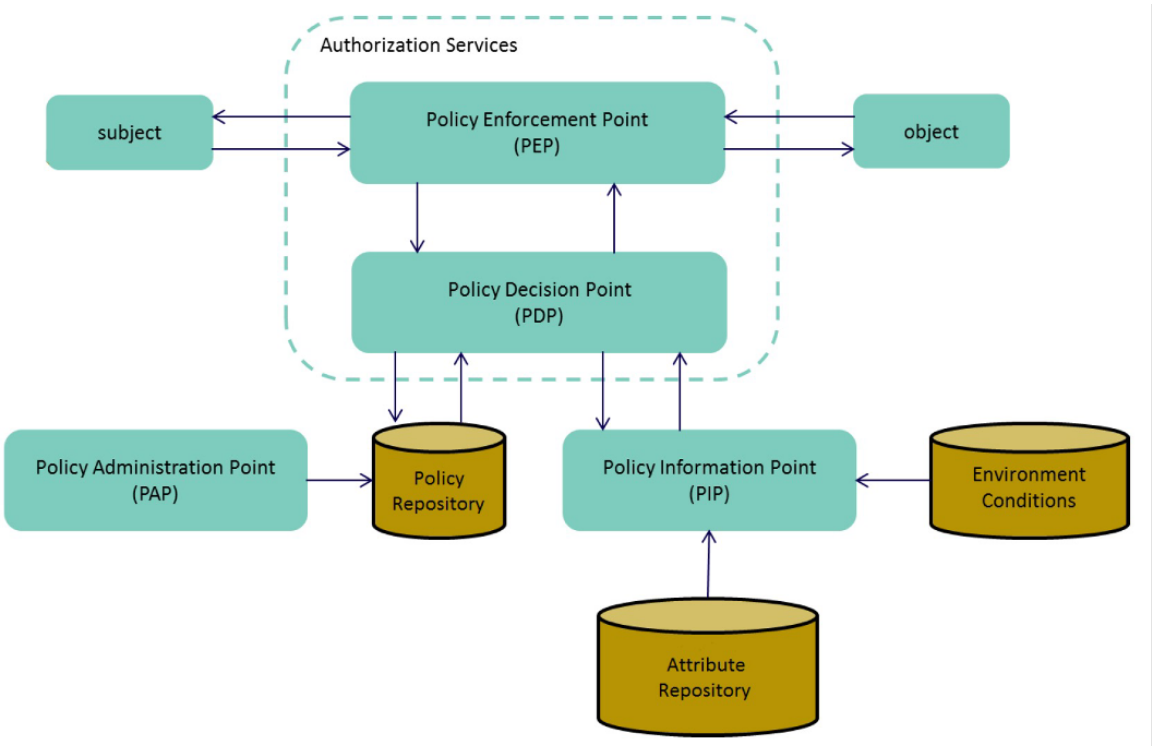


**Figure 2.2:** XACML Reference Architecture [HF+14]

The architecture introduces four components, also referred to as functional points. In this thesis, the term PXP is used to refer to these functional points. These points are the Policy Enforcement Point (PEP), Policy Decision Point (PDP), Policy Administration Point (PAP), and Policy Information Point (PIP). The PAP is responsible for writing policies and making them available to the PDP. Therefore, the policies are stored in a policy repository. The PEP is responsible for enforcing the authorization decisions. To accomplish this, the PEP must intercept the request performed by a subject on an object and forward the request to the PDP. The PDP is responsible for evaluating if the request is allowed or denied. Therefore, the PDP must access the respective authorization policies from the policy repository. In addition, the PDP might require additional attributes for the subject, object, or the environment. These attributes can be retrieved by accessing the PIP. The PIP has access to an attribute repository and environment conditions. The result of the evaluation performed by the PDP is returned to the PEP. If the PDP decided to allow the request, PEP allows the subject to access the object. Otherwise, the request is denied.

### 2.5.3 Policy Languages

XACML is an XML-based markup language used to specify authorization policies [OAS-XAC]. As mentioned in Section 2.5.1, XACML is often referred to as an ABAC model, as it allows creating fine-grained authorization policies using attributes. The main components are rule, policy, and policy set. The rule is the elementary unit of a policy. Among others, a rule has a target, a condition, and an effect. One policy comprises one or multiple rules. A policy set is a set of policies that are applied to a target.

Listing 2.2 presents an example policy written in XACML. The policy has a *PolicyId* (line 2) and a description (line 3). In addition, the policy defines a rule combining algorithm, which specifies how the results of the rules are to be interpreted (e.g., use of logical and/or). The policy only contains one rule (lines 4 to 18). The rule also has an *RuleId* and a description. The rule evaluates if the subject has an email address containing *med.example.com*. The data types must be explicitly declared. As can be seen in the rather simple example presented in Listing 2.2 the policy is specified using an XML markup language does not improve readability. Thus, the Abbreviated Language for Authorization(ALFA) has been developed by Axiomatics and donated to OASIS [OAS-ALF]. ALFA can be mapped into XACML. It improves readability and allows handling authorization policies more like source code.

Another language is provided by the Casbin framework, which is an open-source library supporting multiple access control models, including ABAC [CO-Doc]. Casbin is supported by multiple programming languages and allows specifying authorization policies as source code. This allows Casbin to be highly flexible, as it can be embedded into source code, externally enforced by a PDP, or integrated into Kubernetes as a middleware.

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy ... PolicyId="urn:oasis:names:tc:xacml:3.0:example:SimplePolicy1"
    Version="1.0" RuleCombiningAlgId="identifier:rule-combining-algorithm:
    deny-overrides">
  <Description> Medi Corp access control policy </Description>
  <Rule RuleId= "urn:oasis:names:tc:xacml:3.0:example:SimpleRule1"  Effect
      ="Permit">
    <Description>
            Any subject with an e-mail name in the med.example.com domain
                can perform any action on any resource.
      </Description>
    <Target>
      <AnyOf>
        <AllOf>
          <Match MatchId="...:function:rfc822Name-match">
            <AttributeValue DataType="...string">med.example.com</
                AttributeValue>
            <AttributeDesignator MustBePresent="false" Category="..:
                subject-category:access-688 subject" AttributeId="...:
                subject:subject-id" DataType="...:data-type:rfc822Name"/>
          </Match>
        </AllOf>
      </AnyOf>
    </Target>
  </Rule>
</Policy>
```

**Listing 2.2:** Example XACML Policy [OAS-XAC]

### 2.5.4 Open Policy Agent

Open Policy Agent (OPA) is an open-source policy engine developed by Styra [OP-Do]. The project has been developed for cloud native services and is a graduated project by Cloud Native Computing Foundation (CNCF). Open Policy Agent (OPA) includes a declarative policy language, Rego, which is inspired by the Datalog query language and allows specifying policies as code.

Listing 2.3 presents a Rego policy with the same logic as the XACML policy presented in Listing 2.2. Authorization policies can be structured into allow statements. Multiple boolean rules can be included in an allow statement. All rules must evaluate to true for the policy to evaluate to true (logical and). Rego allows structuring Rules into different packages and files. This can reduce code duplication.

Furthermore, Rego and OPA provide an integrated test framework that allows creating unit tests for policies and rules.

```
1 allow if {
2     "med.example.com" in input.subject.email
3 }
```

**Listing 2.3:** Example Rego Policy

OPA can be used as a PDP which receives the request from a PEP. The content of the request is referred to as input and is structured in the JSON format. The input (i.e., request) is then queried against the Rego policies. If one Rego policy evaluates to true, the request is allowed. Depending on the requirements, OPA can be extended to provide additional functionality (e.g., database access see Section 7.3). Since OPA has been developed for cloud native environments, OPA instances are lightweight and can be used as a sidecar that can be quickly scaled up and down. Due to the cloud native context, Open Policy Agent (OPA) and the Rego policy language have been selected for this thesis. However, the concepts provided in this thesis are language-agnostic and thus also applicable to other policy languages.

## 2.6 Zero Trust

Zero trust and the zero trust architecture are a recent trend in software engineering [TU+21]. The goal of zero trust is to remove implicit trust in an IT system. Implicit trust occurs, e.g., where access is granted based on physical properties such as the same access to a corporate network or deployment on the same physical compute node.

A special NIST publication on zero trust by Rose et al. specifies zero trust and the zero trust architecture as the following:

> ***Zero trust (ZT)*** *provides a collection of concepts and ideas designed to minimize uncertainty in enforcing accurate, least privilege per-request access decisions in information systems and services in the face of a network viewed as compromised.*
> ***Zero trust architecture (ZTA)*** *is an enterprise's cybersecurity plan that utilizes zero trust concepts and encompasses component relationships, workflow planning, and access policies. Therefore, a zero trust enterprise is the network infrastructure (physical and virtual) and operational policies that are in place for an enterprise as a product of a zero trust architecture plan.* (Rose et al. [RB+20, p. 13])

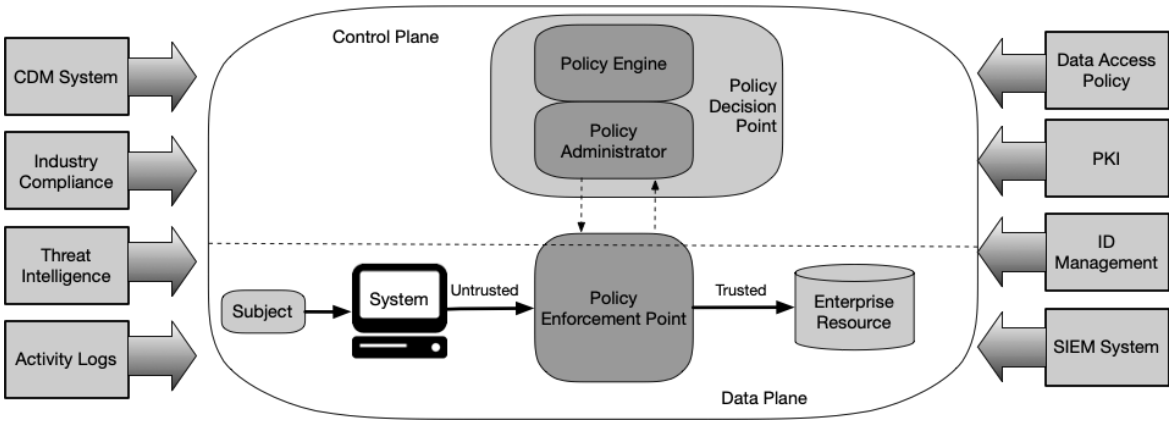


**Figure 2.3:** Zero Trust Architecture by [RB+20, p.18]

The goal of zero trust is the consequent enforcement of the principle of least privilege, following the principle "*never trust, always verify*" [BO+21, p. 2]. Figure 2.3 depicts an idealized overview of the zero trust architecture by Rose et al. The center contains the components of the zero trust architecture. The architecture is divided into a control plane and a data plane. In the data plane, a subject tries to access a resource using a system. The PEP is responsible for "*enabling, monitoring, and eventually terminating connections*" [RB+20, p. 10]. The PDP is located in the control plane. it consists of a policy engine and a policy administrator. The policy engine is responsible for creating a decision granting or denying access to the resource. The policy administrator is responsible for the creation or termination of the connection to the enterprise resource. If the request is allowed, the policy administrator creates a session-specific authentication used by the client to access the resource. The policy administrator will subsequently tell the PEP to allow or deny the request. The policy engine and policy administrator can also be realized by a single component.

When comparing the proposed zero trust architecture with the XACML reference architecture, an overlap can be noticed. By applying the ABAC reference architecture to a microservice-based application, the introduction of the zero trust architecture is supported. This is further elaborated in Chapter 7.

### 2.6.1 Identity Propagation

The concept of identity propagation can be used to support the zero trust architecture. Identity propagation distributes the credentials of a user through the application. As Chandramouli et al. note, "*all application traffic should carry end user credentials [. . . ]*" [CB+21, p. 29]. While the concept has also been proposed for SOA [PM+09], Yarygina and Bagge identified identity propagation as an emerging security practice for the microservice architecture [YB18]. The term identity propagation is also used synonymously with the term principal delegation [YB18]. However, this thesis uses

the term identity propagation. The distribution of user credentials creates an additional angle of attack. For instance, a credential can be stolen and misused by a compromised microservice. To counteract this problem, credentials should have a limited lifetime [YB18] or be exchanged with a limited application-internal credential [Ch19]. JWTs are often used for the propagation through the microservice architecture. These JWTs can be exchanged through an application-internal token to limit the damage of a stolen access token [NJ+18]. However, identity propagation does not influence or modify access rights. This can be provided by the concept of delegation, which is "*[. . . ] a frequently desired access control feature that allows one subject to temporarily delegate their access rights to a more junior (in terms of access rights) subject*" [SO17, p. 29]. We consider the aspect of delegation as future work of this thesis, as Vijayalakshmi and Jayalakshmi identify it as a challenge in ABAC [VJ22].

# 3 State of the Art

This chapter elaborates on the state of the art related to this thesis. To assess the literature in the context of the introduced research questions (see Section 1.3), Section 3.1 creates a requirements catalog. The catalog contains requirements that should be addressed to enable policy-driven authorization in a microservice-based application. Therefore, Section 3.2 introduces and assesses the literature from the state of the art based on the requirements catalog. Finally, based on the assessment, the research gaps addressed by this thesis are introduced in Section 3.3.

## 3.1 Requirements Catalog

The requirements for policy-driven authorization in microservice-based applications are established based on the research questions described in Section 1.3. For every research question, a requirement is created. The requirements R1 to R4 elaborate on the development of authorization policies in the context of an engineering process. The development of authorization should be considered throughout all development phases (R1). This includes the definition of authorization requirements to capture what must be authorized (R2). In addition, the use of fine-grained authorization is a key requirement to perform extensive authorization decisions (R3). In the context of microservice-based applications, the consideration of Service-to-Service (S2S) authorization is an important aspect to remove implicit trust among microservices (R4). Requirements R5 and R6 target the architectural requirements to include components necessary for authorization in the microservice architecture. The authorization should be performed externalized from the microservice (R5) and decentralized in the overall architecture of the microservice-based application (R6).

### R1 - Embedding Authorization Into Development

Software engineering can be structured into different phases. A typical classification uses the phases of (requirements) analysis, design, implementation and test, and deployment and operations. These phases can be organized in various ways. A classical approach is the waterfall model [Ro70]. Modern approaches are more iterative and agile. However, the phases remain the same. To create sustainable and understandable authorization, the aspect of authorization should be present in every development phase [KK+21]. This requires modifications to the respective phases, including guidelines for

developers. Each development phase creates artifacts (e.g., requirement specification, class diagrams) which must be aligned with authorization artifacts. The integration of application development and authorization should be established across all phases and conducted in parallel (see RQ1).

### R2 - Definition of Authorization Requirements

To create software, a clear set of requirements must be defined. As described before, these requirements can be classified as functional or non-functional. Functional requirements must be implemented to enable users of a software to accomplish their respective tasks. A non-functional requirement can be classified as an attribute of a constraint on a system [Gl07]. Security in an IT context is classified as a non-functional requirement (e.g., by ISO 25010). Since authorization can be seen as a part of security [Fi03], authorization requirements can be categorized as non-functional requirements. To integrate authorization into the development of microservice-based applications, the definition of authorization requirements is essential. Otherwise, there is no dedicated knowledge of what should be authorized under a given set of circumstances. Thus, a structured approach towards the elicitation of authorization requirements is required during the requirements analysis (see RQ2).

### R3 - Fine-Grained Authorization

Authorization can be divided into coarse-grained authorization and fine-grained authorization [Ki15]. While coarse-grained authorization would grant access to a microservice itself, fine-grained authorization grants access to a specific resource provided by a microservice depending on the current context. For example, if a subject performs an action to an object, the access decision is evaluated based on, e.g., ownership, time, or age of the subject. To perform such fine-grained authorization decisions, Attribute Based Access Control (ABAC) can be used [AQ+18; AC22]. The use of ABAC for microservice-based applications has also been proposed by Yarygina and Bagge [YB18]. The integration of a strong fine-grained authorization mechanism is a key requirement for the microservice architecture. Thus, this thesis assumes fine-grained authorization as a requirement for authorization in a microservice-based application (see RQ3).

### R4 - Service-to-Service Authorization

A microservice-based application consists of multiple microservices that must communicate with each other to provide the necessary business functionalities. Besides the authorization of requests performed by users, the authorization of requests performed among microservices is a challenge [AC22]. The communication between microservices is often regarded as secure due to the close physical distance between the deployed services. For example, in a Kubernetes context, the communication of

microservices in a single namespace might be considered secure. However, this implicit trust leads to security issues and is regarded as a security smell [PS+21]. If no authorization among microservices is established and a single microservice is compromised, the microservice can access every other microservice. This problem is also known as the confused deputy problem [Ha88]. A countermeasure is to remove the implicit trust by introducing an architecture that requires authorization for every request that is performed among microservices. This is also known as zero trust [TU+21]. The authorization between microservices is hence regarded as a requirement towards policy-driven authorization in microservice-based applications (see RQ4).

#### R5 - Externalized Authorization

In monolithic applications, the application itself commonly handles authentication and authorization [Ne15]. Evaluating authorization decisions in the microservice code can contradict the loose coupling of the microservice architecture. If an aspect regarding authorization changes, the microservice implementation must be modified, tested, and deployed. This reduces the reusability and maintainability of the microservice. Thus, to perform fine-grained authorization decisions, the authorization logic should be performed externally. In this context, externally means the evaluation outside the microservice's code base, which keeps the loose coupling and creates a higher degree of re-usability, since authorization policies and authorization components can be replaced. The externalized authorization is therefore considered a requirement (see RQ5).

#### R6 - Decentralized Authorization

In a distributed microservice architecture, performing authorization at a central point contradicts the core principle of loosely coupled microservices. Thus, centralized authorization in the microservice architecture is regarded as a security smell [PS+21]. If a centralized authorization server handles all authorization requests, it creates a single point of failure. Additionally, if a central authorization server is compromised, the microservice-based application is compromised as a whole. For authorization in microservice-based applications, decentralization of authorization is regarded a requirement (see RQ6).

## 3.2 Assessment of the State of the Art

The following sections present the state-of-the-art literature. Figure 3.1 provides an overview of the selected literature. Two publications consider the aspect of authorization in microservice-based applications in general [NJ+18; SH+21]. Since a goal of this thesis is authorization using policies, four

more publications are selected considering the engineering of said authorization policies. The state-of-the-art on developing authorization policies can be categorized into top-down policy engineering and bottom-up policy engineering approaches [DM+18]. The top-down approaches focus on the authorization policy creation based on natural language documents ([BG+17; XP+12]). The bottom-up approaches create authorization policies based on existing data such as logs or source code ([XZ+23; LC+21]).

While the aspect of automating the creation of authorization policies is not considered in this thesis (see Premise 6, Section 1.5.6), the generation of policies is an active research field. For this reason, three of the six sources examined ([XP+12; XZ+23; LC+21]) also deal with the aspect of the automated creation of authorization policies. However, the fundamental ideas of the publications still hold for the manual generation of authorization policies. At the time of writing this thesis, the generation of authorization policies using a Large Language Model (LLM) has only been proposed by Martinelli et al. [MM+24]. Employing LLMs in the generation of authorization policies will likely lead to better results due to the improved processing of natural language documents (e.g., requirements). However, the automation is considered as future work (see Section 9.2).

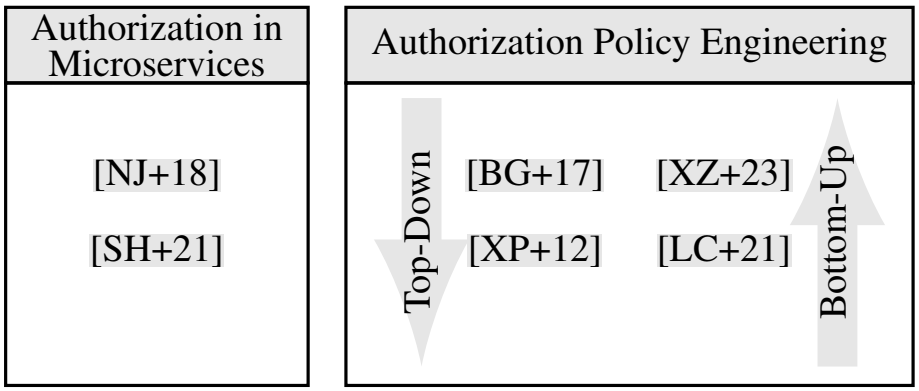


**Figure 3.1:** Overview of the Selected Literature

In the following, the focus is set on presenting the core contributions of the respective publications. In addition, the publications are assessed regarding the requirements catalog created in the previous section. For each publication category depicted in Figure 3.1, a brief overview of further literature is provided.

### 3.2.1 Authorization in Microservice

#### Nehme et al.: Fine-Grained Access Control for Microservices [NJ+18]

Nehme et al. [NJ+18] proposed one of the first approaches towards fine-grained authorization in microservice-based applications. The authors assume that there are two kinds of microservices. First,

primitive resource microservices, which only provide primitive functionality such as Create Read Update Delete (CRUD) operations on assets (e.g., personal data). These microservices do not contain any authorization logic and are protected by a local gateway. Each resource microservice has its own local gateway that can enforce security policies. Second, consumer microservices, which provide (business) functionality to a user. To do so, the consumer microservices access assets provided by the resource microservices.

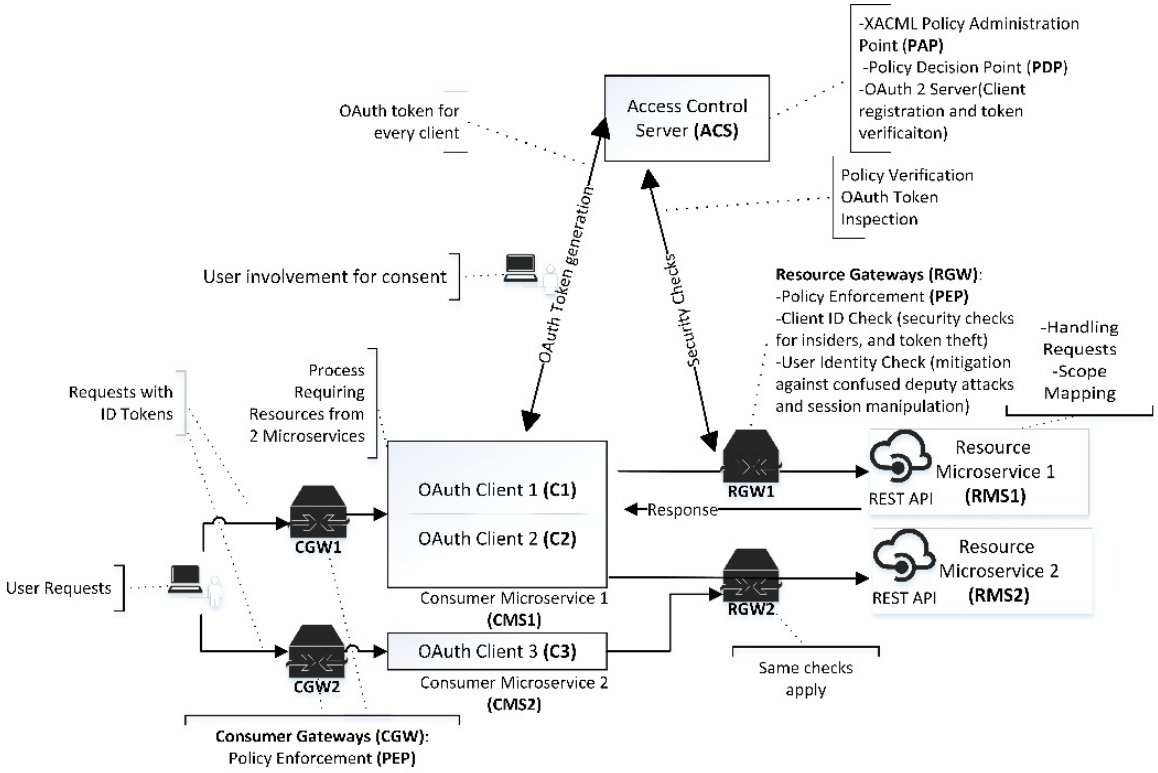


**Figure 3.2:** Security Architecture Proposed by Nehme et al. [NJ+18]

The authors propose a security architecture that addresses fine-granular access policies, separates control between user and service providers, and verifies authenticity of consumers. Figure 3.2 presents the security architecture. The central component of the architecture is an Access Control Server (ACS), which provides the functionality of an eXtensible Access Control Markup Language (XACML) server (i.e., PAP and PDP) and an OAuth 2.0 server allowing to register new clients and verify tokens. Similar to the resource microservices, each consumer microservice has its dedicated consumer gateway which act as a Policy Enforcement Point (PEP). The consumer gateway intercepts every user request and extracts the ID token. Subsequently, the consumer gateway performs a policy check for user access at the ACS. If the check is successful, the request is forwarded to the consumer microservice. For each connection from a consumer microservice to a resource microservice, a dedicated OAuth client is registered at the ACS. To access the data from a resource microservice, an OAuth access token is requested by the consumer microservice from the ACS. The user must consent to the access

scope defined by the consumer microservice. If successful, the consumer microservice forwards the generated access token and the user's ID token to the resource microservice. The request is intercepted by the resource gateway, which performs several checks: First, the client ID is checked from the access token. Second, the user identity is checked from the ID token. Third, the authorization policy is checked for the user access. If one of the checks fails, an incident is reported. Otherwise, the request is forwarded including the scope and the user ID. The resource microservice responds with the requested data to the consumer microservice.

With the proposed architecture, Nehme et al. aim to limit the power of an access token between consumer microservices and resource microservice as they have a short lifetime. In addition, the authors address the confused deputy problem by requiring the user's ID token in combination with the access token. Thus, a consumer microservice with a valid access token cannot access assets for another user from a resource microservice.

**Assessment** The security architecture proposed by Nehme et al. introduces an approach towards fine-grained authorization in the microservice architecture. The authors employ OAuth 2.0 and XACML in a centralized access control server to enforce authorization decisions. However, the authors do not address how the architecture is embedded in the application development. Thus, requirement R1 is not met. The authors create security requirements towards the realization of fine-grained authorization in their security architecture. These requirements should be considered when enforcing the authorization policies in a microservice-based application which is introduced in Chapter 7. Nehme et al. do not address the definition of authorization requirements. Therefore, requirement R2 is not addressed.

With the use of XACML and OAuth 2.0, the authors provide fine-granular authorization decisions. Hence, requirement R3 is satisfied. However, the concrete use of XACML is not described in detail. The complexity of using XACML and OAuth in combination with the ACS, might be avoidable if XACML was fully used. The authors address the authorization between microservices, complying with R4. The authors additionally consider the problems of powerful access token theft and the confused deputy problem. With the proposed security architecture, the implicit trust between microservices is removed.

Generally, the authorization decisions are performed externally from the consumer microservice and the resource microservice. The authors explicitly state that the resource microservices are released from the logic required for authorization. However, since the resource microservice receives the user's ID token and the request scope, it is unclear if additional authorization checks are made in the resource microservice. This also applies to the consumer microservices. The authors do not state if the ID tokens are only used for the later authorization with the resources microservices or for additional authorization checks. Assuming there are additional authorization mechanisms /

checks embedded in the microservice code, the requirement R5 is only half fulfilled. R6 requires decentralized authorization. Due to the central access control server of the security architecture, the aspect of decentralization is not met. If the ACS introduced in the security architecture fails or is compromised, a correct authorization can no longer be guaranteed. Hence, the ACS presents a single point of failure.

### Sauwens et al.: ThunQ: A Distributed and Deep Authorization Middleware for Early and Lazy Policy Enforcement in Microservice Applications [SH+21]

Sauwens et al. [SH+21] provide an authorization middleware for microservice-based applications. Their solution focuses on authorization as early as possible in the microservice-based application, as well as lazy authorization. This means that authorization decisions are performed in the relevant data context (i.e., microservice). Hence, the authorization middleware is distributed among a microservice-based application. Each microservice must include the authorization middleware. The authors consider a microservice-based application that is used by multiple tenants. The tenants interact with the microservices through an API gateway, which forwards requests to respective microservices. Among microservices, requests can be performed to exchange data. Each microservice has its database to persist data.

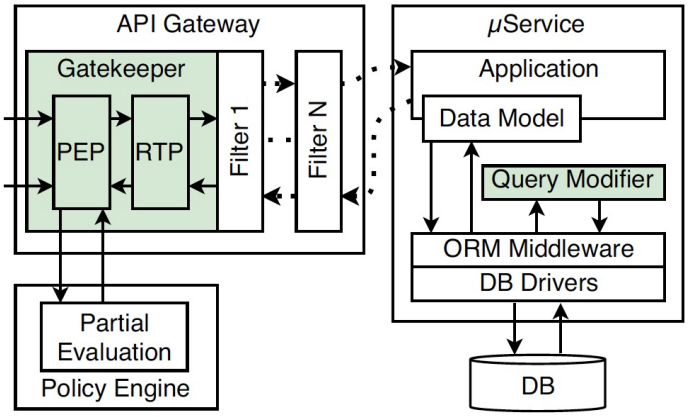


**Figure 3.3:** ThunQ's Components by Sauwens et al. [SH+21]

The solution proposed by Sauwens et al. is called ThunQ. The components are depicted in Figure 3.3. The entry point for the microservice-based application is the API gateway, which is extended by the Gatekeeper component. The Gatekeeper is implemented as a filter for the Spring Cloud Gateway and contains a PEP and a Request Transformation Point (RTP). The PEP is a modified XACML PEP that forwards the requests to a policy engine. The policy engine is provided by Open Policy Agent (OPA). OPA partially evaluates the policy and returns the result to the PEP. The partial evaluation only evaluates the rules where the information is available at that point. The results of partial evaluation are

```
1 allow {                                allow {
2  user.tenant == "insurer"                doc.tenant_id == 67
3  doc.tenant_id == user.tenant_id         doc.employee_id == 52
4  user.role == "account_manager"        }
5  doc.employee_id == user.id
6 }
```

**Listing 3.1:** Exemplary Partial Policy Evaluation [SH+21]

depicted in Listing 3.1. The left allow statement contains four rules (lines 2 to 5). The rules in lines 2 and 4 can directly be evaluated to true, as the required user attribute data is available. The attributes *tenant_id* and *employee_id* of the *doc* object are unavailable. Hence, the partial result displayed on the right half of Listing 3.1 is returned to the PEP to be evaluated by a microservice.

The RTP uses the result of the partial evaluation and creates a thunk. A thunk is a key values structure which contains a URL path selector and a query modifier. Listing 3.2 shows an example thunk in lines 1 to 3. The path selector defines which residual policy should be evaluated by a database query. The query modifier contains boolean expressions which are the result of the partial evaluation. The thunks are then added to the request that is forwarded to the microservices.

```
1 {
2     "/accountStates /*":" doc.tenant_id = 67 && doc.employee_id = 42", ..
3 }
4 ---
5 SELECT *                      SELECT *
6 FROM account_states           FROM account_states
7                               WHERE tenant_id = 67 AND employee_id = 42
8                               AND <BoolExpr #...>
```

**Listing 3.2:** Exemplary Thunk and Query Modification [SH+21]

As depicted in Figure 3.3, each microservice contains a query modifier which rewrites the queries that are performed to the records of a microservice database. The query modifier is a plugin of the Object Relational Mapper (ORM) that uses the received thunk to determine the query modifications. If the path selector matches a relevant database object, the boolean expressions are added to the database query. An example is depicted in lines 5 to 8. On the left side, the initial database query is depicted. The right half depicts the modified query, including the boolean expressions from the thunk. If a microservice requires data from another microservice, the thunk is added to the request performed to the other microservice.

**Assessment** The authorization middleware ThunQ proposed by Sauwens et al. provides a solution to evaluate authorization decisions as early as possible. The authors focus on the development of the middleware and do not introduce how to use the middleware systematically or how to create the authorization policies. It is unclear, how the relevant attributes are determined and how the mapping to the query works. Therefore, the requirement R1 is not satisfied. This is also true for R2, the definition of authorization requirements, which are not addressed by Sauwens et al.

The authors use ABAC to define authorization policies using the ThunQ middleware. The authorization decisions are fine-grained by modifying the access to the database records. That is why the requirement R3 can be classified as satisfied.

The primary assumption by the authors is that the microservices operate correctly and can be trusted. Therefore, the authors do not consider the authorization between microservices. If a microservice were to be compromised, the thunk could be manipulated by modifying the query modifier. This might lead to data leakage or SQL injections. The requirement R4 is therefore not fulfilled.

The authorization decisions performed by ThunQ are partly externalized. The initial authorization decision is performed externally by the Open Policy Agent. However, the partially evaluated policies are authorized by the query modifiers which are embedded as a plugin in the microservice code (i.e., database connection). Thus, requirement R5 is half satisfied. While the authorization decisions are enforced decentralized in the microservice-based application using ThunQ, a centralized access control server (i.e., PDP and RTP) is still necessary. This creates a single point of failure. The requirement R6 is considered half met.

### Further Related Work

Complementing the technical realization of authorization, Bánáti et al. proposed a centralized authorization orchestrator for microservices-based applications, which required a dedicated "IAM auth" module in the microservices [BK+18]. The authors used OAuth2.0 and JSON Web Token (JWT) to perform authorization decisions. Miller et al. [MM+21] show an approach to deploy a microservice in a cloud infrastructure. Using Kubernetes to deploy microservices, the authors propose the use of the sidecar pattern [BG+17] to deploy a dedicated API proxy and OPA container next to each microservice. Similar to the publication by Nehme et al. [NJ+18], the proposed approaches do not consider what must be authorized and how to integrate the architectural modifications systematically in the development of a microservice-based application.

### 3.2.2 Top-Down Policy Engineering

#### Brossard et al.: A Systematic Approach to Implementing ABAC [BG+17]

Brossard et al. [BG+17] present a systematic approach to the implementation of ABAC. The authors work for Axiomatics, a company developing policy-based authorization, including the policy language ALFA [OAS-ALF]. They refer to the key elements of ABAC as the attributes, policies, and deployment architecture. The attributes are divided into subject, action, resource, and environment categories. A policy then combines attributes to express positive or negative authorization cases.

The authors identify three challenges that arise with the use of ABAC. First, compared to RBAC user permissions are not directly assigned to a user's role but are the result of a runtime authorization evaluation against policies. Thus, a process is needed for the provisioning of policies and subsequent access reviews. Second, traditional implementations using RBAC lack requirements due to the implementation inside the code of an application. When using ABAC, the authorization requirements must be gathered and managed. The authorization requirements are at the level of a use case or an application and are implemented as authorization policies. Third, the ownership of ABAC authorization policies is unclear. The authorization logic is embedded within the authorization policies, and the development can require application developers or IT/IAM staff. With RBAC, fine-granular authorization is implemented in the code with the responsibility of the application developers, while coarse-grained decisions are performed by a central IAM team.

To address these challenges, Brossard et al. propose a policy lifecycle depicted in Figure 3.4. The first step is the definition of use cases, which specify the functionality and the authorization scope. Based on the use case definition, the authorization requirements are created. An authorization requirement is a natural language statement that defines what should be allowed or forbidden. Subsequently, the attributes required for the authorization policies must be identified. Therefore, the natural language authorization requirements are examined. For each attribute, the authors propose the formulation of various characteristics such as a short name, a namespace (i.e., the logical domain an attribute belongs to), a category (e.g., subject, action, resource, environment), value constraints, or a data type. The attribute definition is followed by the authoring of the authorization policies. In this step, the natural language authorization requirements are broken down into atomic attributes and attribute comparisons using the previously defined attributes. An important aspect is the normalization of attribute values (i.e., upper/lower case). The statements can then be implemented in a policy language (e.g., ALFA, XACML). Then the implemented policies must be tested. The authors propose binary testing, gap analysis, or reverse query testing. Tests should be run every time a policy is modified. The tested policies are deployed to the ABAC architecture, including the PEP and the Policy Decision Point (PDP). The final step is to frequently run access reviews to review if a user has gained or lost access rights.

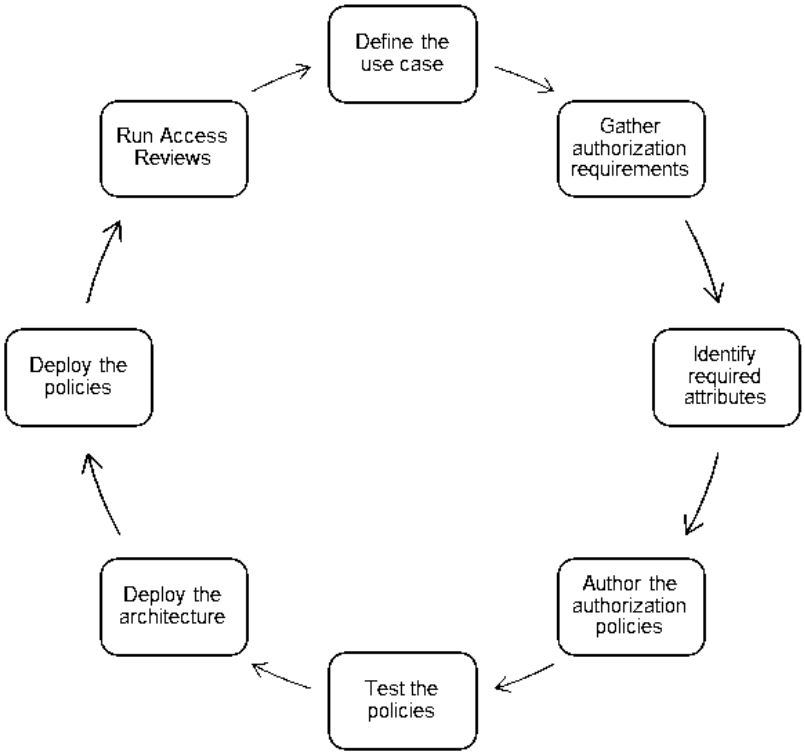


**Figure 3.4:** Authorization Policy Lifecycle by Brossard et al. [BG+17]

An example of the resulting authorization policy is shown in Listing 3.3. The policy is written in the policy language Alfa. The target in line 2 defines the object. Line 3 defines the applied combining algorithm (similar to XACML). The authorization requirement is realized by a rule (lines 5 to 9) that must evaluate to true to allow access. Inside the rule, the attributes are evaluated using atomic comparison operations.

```
1 policy records {
2     target clause object.objectType == "record"
3     apply firstApplicable
4     /** R2 - An employee can view a record in their own department */
5     rule employeesView {
6       target clause user.role == "employee" and action.actionId == "view"
7       condition user.department == record.department
8       permit
9     }
10 }
```

**Listing 3.3:** Exemplary ALFA Authorization Policy [BG+17]

**Assessment** Brossard et al. propose a systematic approach to implementing ABAC. The authors argue that compared to RBAC, ABAC creates new challenges due to its distributed nature. This also includes the organization and the involved stakeholders. In their work, the authors consider the development of the ABAC authorization policies from the identification of the use cases to the deployment of the policies and the operation (i.e., access reviews). The focus of the lifecycle is not strictly set on the development of an application and does not specifically consider development artifacts. While the authors begin with the definition of use cases and the subsequent definition of authorization requirements, they do not consider further software development artifacts. For instance, the step to identify attributes should be performed in the context of the software design. If an identified attribute is different in the software design (e.g., API specification, class diagram), the policy cannot succeed. Hence, the requirement R1 is only partially met.

The authorization requirements proposed by Brossard et al. define what is to be authorized based on the use cases in natural language. The authors do not define a structure for the authorization requirements. In addition, no systematic approach for the definition of the authorization requirements is provided. The requirement R2 is only partially satisfied. With the use of ABAC, the authors follow fine-grained authorization. The requirement R3 is therefore fulfilled.

The aspect of authorization between microservices is not addressed in this paper. Hence, the requirement R4 is not applicable. The ABAC architecture is briefly addressed by Brossard et al. The authors consider a centralized ABAC service which contains the authorization policies. The PDP can be deployed behind a load balancer. The PEP is deployed as an API gateway in different levels. The authorization decision is created and enforced externally from the application. Requirement R5 is therefore fulfilled. Due to the use of a centralized ABAC service, the requirement R6 is considered as not met.

### Xiao et al.: Automated Extraction of Security Policies from Natural-language Software Documents [XP+12]

Xiao et al. [XP+12] develop Text2Policy, one of the first approaches to extract authorization policies (called Access Control Policies (ACP) by the authors) from natural language software documents. More recent approaches improve the performance of Text2Policy. However, the fundamental concepts of Text2Policy remain comparable to newer approaches. For the authors, authorization policies are written in natural language and are a type of non-functional requirement. The authors define two major issues with the definition of the natural language authorization policies. First, the incorrect specification of the policies. This can be a result of the vast amount of complex rules. The non-functional requirements are not explicitly defined. Instead, the information required to define authorization policies is contained in a natural language document or a functional requirement such as use cases. The second identified issue is the incorrect enforcement of specified authorization policies, which

occurs due to the gap between the authorization policies and the implemented system. Therefore, the authors use natural language requirements to extract authorization policies. The advantage of these requirements are the clear and simple structure of the sentences. These requirements can also display different scenarios and steps taken to fulfill them (e.g., use cases).

Figure 3.5 depcits the Text2Policy approach. The approach contains the steps of linguistic analysis, model instance construction, and formal specification, which each create and use intermediate artifacts. First, the linguistic analysis takes the natural language documents as an input and performs a syntactic and semantic analysis. Therefore, a sentence is deconstructed into subjects, verb groups, and objects. For use cases containing several steps, the sentences containing the steps are identified. In addition, the verb groups are assigned to a pre-defined semantic class. Using pattern matching, the sentences which are a natural language authorization policy are identified and annotated with a semantic pattern.

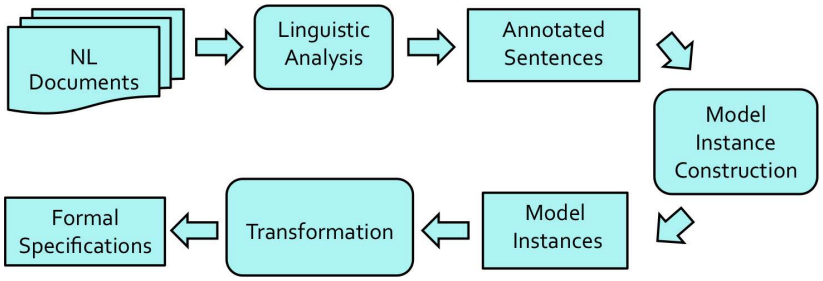


**Figure 3.5:** Text2Policy Approach by [XP+12]

The second step uses the annotated sentences to create a model containing the subject, action, resource elements, and the policy effect (e.g., allow, deny). Therefore, first, the elements are identified from the annotated sentences. Additionally, the policy effect is inferred from the verbs contained in the sentence.

Finally, the model instances are transformed into a formal policy specification. Text2Policy uses XACML to formulate the authorization policies. Listing 3.4 shows a resulting XACML policy. The effect of the policy is defined as *Deny* in line 3. If the policy is evaluated to false, the request is denied. The role of the subject is verified in lines 7 and 8. The subject must maintain the role *HCP* to access the policy. Line 12 as *patient.account* presents the extracted resource. Finally, line 17 defines the allowed action to the resource as *UPDATE*.

Xiao et al. evaluate Text2Policy on the open-source teaching applications iTrust and an internal IBM application. Text2Policy can identify more than 88% of the sentences containing a natural language authorization policies. The accuracy of the extraction of the model elements is more than 81%.

```
<Policy PolicyId="2" RuleCombAlgId="...">
    <Target/>
    <Rule Effect="Deny" RuleId="rule-1">
    <Target>
    <Subjects><Subject>
        <SubjectMatch MatchId="string-equal">
        <AttrValue>HCP</AttrValue>
        <SubjectAttrDesignator AttrId="subject:role"/>
    </SubjectMatch></Subject></Subjects>
    <Resources><Resource>
        <ResourceMatch MatchId="string-equal">
        <AttrValue>patient.account</AttrValue>
        <ResourceAttrDesignator AttrId="resource-id"/>
        </ResourceMatch></Resource></Resources>
    <Actions><Action>
        <ActionMatch MatchId="string-equal">
        <AttrValue>UPDATE</AttrValue>
        <ActionAttriDesignator AttrId="action-id"/>
    </ActionMatch></Action></Actions></Target>
</Rule></Policy>
```

**Listing 3.4:** Exemplary XACML Policy Generated by Text2Policy [XP+12]

**Assessment** Text2Policy introduces a top-down approach to create authorization policies based on natural language documents. The aspect of authorization is not fully embedded into the development of an application by Text2Policy. However, the approach can be classified to starting in the analysis phase. The natural language policies are implemented into authorization policies without incorporating the design artifacts. The authors propose the use of the generated authorization policies as a support for developers. Since the created policies are not complete (e.g., overlooked policies), the missing policies must be manually implemented. Requirement R1 is therefore partly met. The annotated sentences created by the linguistic analysis can be considered an authorization requirement. The sentences stem from the analysis artifacts (e.g., use cases) and could be used to create fine-grained authorization policies. Since the annotated sentences do not follow a similar structure, the requirement R2 is partly fulfilled. The authorization policies created by Text2Policy rely on the subject, the object, and the action. The policies do not contain attributes which would allow fine-granularity. The requirement R3 is thus not satisfied.

The authorization of microservices is not considered by Text2Policy. Therefore, the authorization between microservices has not been considered and R4 is not assessable. While the authors develop XACML policies which would allow for externalized authorization, the authors do not further elaborate on the enforcement of the policies. Therefore, the requirements R5 and R6 are also not applicable.

#### Further Related Work

In the context of top-down policy engineering, automated approaches similar to Text2Policy by Xiao et al. [XP+12] have been proposed. Narouei et al. [NK+17] use Recurrent Neural Networks (RNN) on natural language policy statements to extract policy related information. Similar work has been performed by Alohaly et al. [AT+19], which use Convolutional Neural Networks (CNN) to extract the policy statements. Heaps et al. [HK+21] use the transformer model BERT Large to extract access control information. None of these approaches transform the extracted policy information into a policy language (e.g., XACML). Furthermore, these approaches improve on the performance metrics (e.g., $F_1$ score) but do not deliver perfect results. They can only be considered to support developers with the development of authorization policies. Future approaches will likely make use of the rapid progress that has been made on LLMs such as GPT-4. This has already been proposed by Martinelli et al. [MM+24]. There are also model-driven approaches towards the generation of XACML proposed by Busch et al. [BK+12] or Fatemian et al. [FZ+21]. These approaches define a metamodel, which is subsequently transformed into a XACML policy. However, these approaches still require a software developer to know what must be authorized. In the context of Role Based Access Control (RBAC), Pilipchuk uses business processes specified with Business Process Model and Notation (BPMN) to automate the extraction of access control policies [Pi23].

### 3.2.3 Bottom-Up Policy Engineering

#### Xu et al.: Log2Policy: An Approach to Generate Fine-Grained Access Control Rules for Microservices from Scratch [XZ+23]

Xu et al. [XZ+23] provide Log2Policy, an approach to generate access control policies based on access logs. The generated authorization policies are created for Istio, a service-mesh solution, in which each rule contains the fields from (i.e., source of a request), to (i.e., request target), and when (i.e., conditions of a request) [IA-Do]. Xu et al. argue that the manual generation of fine-grained authorization policies is impractical due to the relationships between microservices in addition to the frequently occurring updates. The foundation for Log2Policy are the access logs of the microservice-based application. It is not further described whether the access logs stem from Istio or the microservices themselves.

The author's threat model assumes that the roll-out of a microservice-based application follows the two phases testing and production. In the testing phase, the microservices are trusted, and the necessary access logs are generated internally by the development teams. To include the user authorization, the authors assume that the user interface is deployed in a front-end microservice, which also creates access logs. In the production phase, the microservices may be compromised by an attacker, allowing to send arbitrary requests.

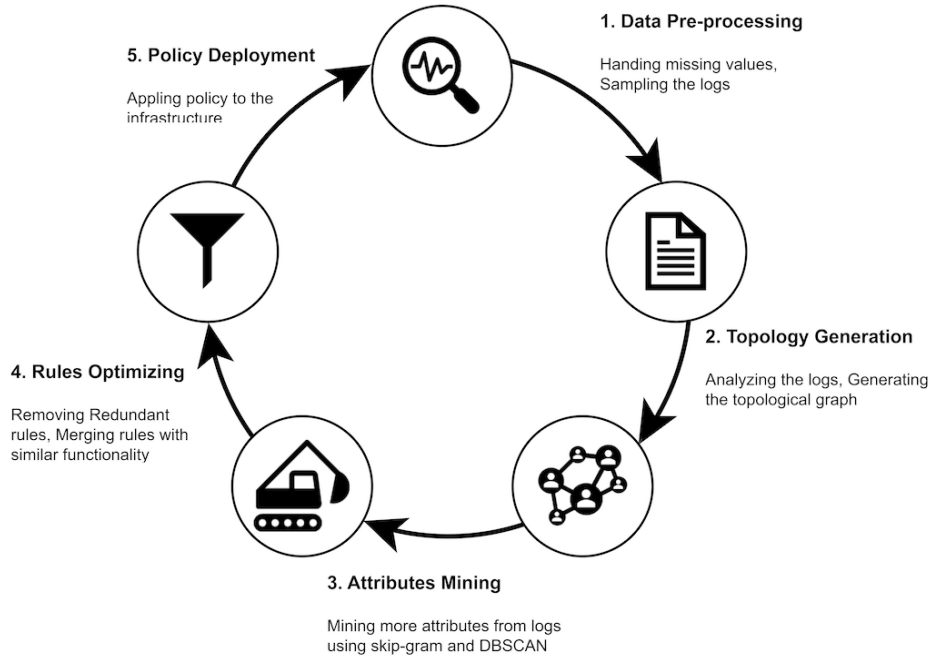


**Figure 3.6:** Proposed Approach by Xu et al. [XZ+23]

To create the authorization policies, the authors propose an approach containing five steps, which is depicted in Figure 3.6. In the first step, the log entries are sampled. The authors argue, that analyzing all logs might take a long time, reducing the practicality of the approach. The sampling rate must be selected adequately to minimize the risk of missing a log entry. Additionally, missing values, e.g., occurring on TCP connections, are replaced by the value NULL. Based on the selected access log, a set of traffic attribute tuples is generated. The content of the traffic attribute tuples are the requester attributes, object attributes, and session attributes. Additionally, the tuples consist of the source microservices and its version, the target microservice including a port, the protocol, the HTTP method, and the HTTP path. Using the traffic attribute tuples, the graph is created as an intermediate step to structure the requests. The graph contains four elements and is structured as $G = (N_s, N_v, E_v, E_i)$. $N_s$ contains the microservices, which can have multiple versions that are stored in $N_v$. The set $E_v$ represents the edges between a microservice and its versions. The set $E_i$ contains the invocation between microservices. The attributes from the traffic attribute tuples are added to the respective invocation edges.

Based on the defined graph structure, the attributes required for fine-grained authorization are mined. The authors assume, that all the attributes relevant for authorization can be extracted from the access logs. They focus in particular on the HTTP path variable, arguing that the relevant attributes are encoded in the HTTP path. To detect these attributes, a combination of algorithms (e.g., DBSCAN) is used. After applying the algorithms to the request presented in Listing 3.5, the location in which the

```
1 GET /carts/<customerID>/items
```

**Listing 3.5:** Extracting Attributes From HTTP Path [XZ+23]

variable *customerID* occurs in the path variable is determined. For the subsequently created policy, the location in which the *customerID* occurs is replaced by a wild card (i.e., *). Hence, all authorization policies created by the algorithm focus on the matching of the HTTP path and introducing wildcards. In the final step, the policies are optimized based on the topology graph. This reduces the amount of authorization policies that are based on a similar HTTP path. As a result, fewer policies need to be evaluated and latency can be reduced. To accommodate frequent microservice changes, Xu et al. address the update process for policies in case of the addition of a new service (version) or the removal of a service (version). Log2Policy only requires updating the graph and to analyze new logs or remove the policies associated with a removed service (version).

The authors evaluate Log2Policy by applying the approach to a set of five popular microservice-based applications. To generate an initial set of logs, a set of test scripts provided by the applications is used. With a sampling rate of 10%, the authors can identify 100% of the requests performed by the applications and generate the respective policies. Reducing the sampling rate reduces the identification of the possible requests, depending on the considered application.

**Assessment** With Log2Policy, Xu et al. provide an approach to generate authorization policies based on the logs created during the operation of a microservice-based application. Since the authors intend the use of Log2Policy before the production stage of an application, other development artifacts are not considered for the authorization policies. This is intended by the authors, as they argue that the manual consideration is too complex. Hence, the requirement R1 is only partially satisfied. With Log2Policy relying solely on log files, the requirement R2 is not applicable.

Additionally, the authors find that not all logs should be considered, as this increases the execution time of Log2Policy. Hence, only a sample of the existing logs (e.g., 10%) is to be selected. This leads to two questions: First, how to select an adequate sample size to not miss an API request between microservices. Second, what happens if a request is not contained in the initial log collection, e.g., it has not been performed during the testing stage. The collection of the authorization policies might become incomplete. Moreover, while the authors claim their authorization policies to be fine-grained, the primary source for the attributes is the HTTP path contained in the performed request. Further attributes are not considered. This also has limitations on the API paradigms that can be used. For example, gRPC uses the HTTP path to structure gRPC services and their respective RPCs. The attributes are inside the binary encoded gRPC messages, which are inaccessible to the policy. In

addition, in the final authorization policies, the authors replace the attributes through wildcards. This effectively reduces the policy to path matching. Therefore, the policies are rather coarse-grained, and we consider requirement R3 to be partially met.

If the correct log sample is selected, Log2Policy can detect all interactions between the microservices of an application. The created authorization policies then enforce authorization on the requests between microservices, complying with requirement R4. However, if an interaction between a microservice is missed, there is no authorization policy. In that case, a default policy denying all requests apart from the detected should be put in place.

The microservices require additional authorization checks (in code) as the authorization policies are not fine-grained enough. For instance, if a microservice is compromised and there is no additional authorization, it could retrieve arbitrary data on an allowed path. The requirement R5 is only partially met. The authorization policies are enforced by Istio in the sidecar of every microservice. The aspect of decentralization (R6) can be considered as fulfilled.

### Li et al.: Automatic Policy Generation for Inter-Service Access Control of Microservices [LC+21]

Li et al. [LC+21] introduce AutoArmor, a tool for the automatic generation of S2S access control policies between microservices in a Kubernetes cluster or an Istio service mesh. The authors use the source code of microservices to generate the authorization policies. The authors argue that manual creation and configuration is error-prone and too time-consuming.

To generate the authorization policies, AutoArmor performs two steps: First, the source code is statically analyzed and a request extraction mechanism extracts the requests performed to other microservices. A request is considered to be a HTTP path including the respective HTTP operation (including gRPC) or a TCP connection (e.g., used for databases). Second, the extracted requests are modeled in a graph structure called a permission graph. The permission graph $G = (N_s, N_v, E_b, E_r)$ considers two kinds of nodes and edges. The service nodes $N_s$ are created for each microservice, while the version nodes $N_v$ are created for each microservice version (e.v., V1.0, V2.0) contained in $N_s$. The edges contained in $E_b$ connect the version nodes to the service node. The allowed requests are modeled as edges in $E_r$ either between service nodes or version nodes. A request edge contains the request type, the path, the method, and the port. Based on the permission graph, the authorization policies are generated. The authors consider a problem they call "over-authorization". If a microservice had $n$ versions and every version of a microservice in $N_v$ has a requesting edge to another microservice, this would result in $n$ authorization policies. The authors argue that the resulting $n$ authorization policies are redundant and lead to higher costs (i.e., latency) when evaluating the policies in an API

proxy. Hence, AutoArmor reduces the authorization policies to a single policy, if all versions of a microservice access another microservice.

A resulting authorization policy for a medical application is presented in Listing 3.6. The policy is a Kubernetes manifest file for Istio's service mesh extension [IA-Do]. The policy defines the source as the *diagnosis service* (line 8), the target service path (line 11), and the HTTP method (line 12). In this case, the version header *v1* is required to perform the request (lines 14 and 15). If all versions of the requesting service are allowed to access the path, the respective restriction can be removed.

```
1  apiVersion: security.istio.io/v1beta1
2  kind: AuthorizationPolicy
3  spec:
4      ...
5      rules:
6      - from:
7          - source:
8              principals:["cluster.local/ns/default/sa/diagnosis"]
9          to:
10         - operation:
11             paths: ["/patients/*"]
12             methods: ["GET"]
13             when:
14             - key: request.headers[version]
15                 values: ["v1"]
```

**Listing 3.6:** Exemplary Istio Authorization Policy [LC+21]

Figure 3.7 presents the proposed architecture to automate the process of generating the authorization policies. A Continuous Integration / Continuous Deployment (CI/CD) pipeline continuously performs static code analysis when the source code is submitted to a Git repository. As a result, a manifest file containing the S2S requests is created. The graph structure takes the manifest file as input, and the policy generator uses it to generate the Istio authorization policies. The Istio control plane applies the authorization policies, automatically synchronizing them with the Kubernetes worker nodes.

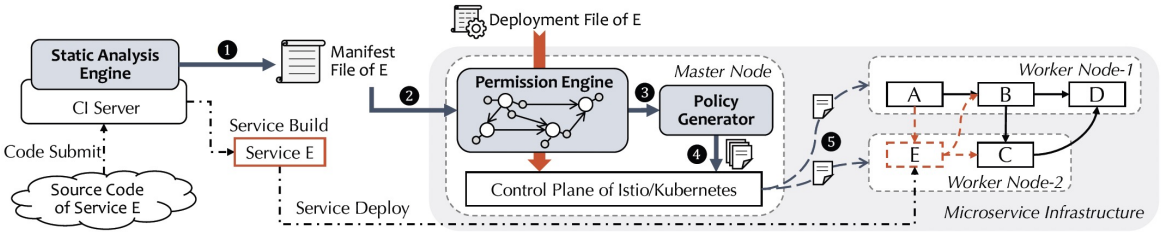


**Figure 3.7:** Architecture of AutoArmor from Li et al. [LC+21]

**Assessment** The authors present an automated approach towards the generation of authorization policies for S2S requests. The approach begins in the implementation phase. The foundation for the authorization policy is the source code. It does not consider artifacts created during the analysis or design phase. Hence, the requirement R1 is only partially addressed and requirement R2 cannot be assessed. While the AutoAmor approach introduces automation, simplicity, and less manual work for a developer, the approach also introduces drawbacks.

The granularity of the authorization policies is restricted to the information available in the code. This only allows defining rather simple authorization policies that do not include the context in which the S2S request is performed. The attributes that are contained in a S2S request are not considered in the AutoArmor approach. For example, a compromised microservice can access the data of all patients provided by another microservice, even though this should be restricted to the patients treated by the doctor initiating the S2S request. Therefore, the granularity of the authorization policies is not fine-granular, e.g., compared to ABAC. This leads to the requirement R3 to be only partially satisfied. In addition, the reliance on source code prohibits the (re-)use of external microservices without access to the source code.

As AutoArmor focuses on the authorization between microservices, the requirement R4 is fulfilled. In addition, AutoArmor requires a service mesh with Istio to enforce authorization decisions. Since Istio uses an extended version of Envoy to run as a sidecar in the Kubernetes deployments, the authorization policies are enforced outside the microservice in the Envoy sidecar. This fulfills requirement R6. However, similar to Xu et al. [XZ+23], since the authorization policies are rather coarse-grained, additional authorization mechanisms in the microservice are necessary. Therefore, requirement R5 is only half met.

#### Further Related Work

In the bottom-up engineering of authorization policies, Cotrini et al. [CW+18] present another approach to mine ABAC rules from sparse access logs, focusing on an organizational setting in which this data is available. In the context of the development of a new application, the lack of access logs from a production system remains a problem. When migrating to ABAC, Talukdar et al. [TB+17] propose an approach to mine ABAC policies based on an existing access control mechanism (e.g., RBAC, ACL).

## 3.3 Research Gaps

The research gaps addressed by this thesis result from the assessment of the existing literature based on the established requirements. Table 3.1 provides an overview of the assessment. A completely

fulfilled requirement is marked by ●. If a requirement is only partly met, the symbol ◐ is used. A requirement that is not fulfilled by the publication is identified with the symbol ○. If a requirement is not applicable to a publication, the character / is used.

| | R1 - Embedding Authorization Into Development | R2 - Definition of Authorization Requirements | R3 - Fine-Grained Authorization | R4 - Service-to-Service Authorization | R5 - Externalized Authorization | R6 - Decentralized Authorization |
|---|---|---|---|---|---|---|
| [NJ+18] Fine-Grained Access Control for Microservices | ○ | / | ● | ● | ◐ | ○ |
| [SH+21] ThunQ: A Distributed and Deep Authorization Middleware for Early and Lazy Policy Enforcement in Microservice Applications | ○ | / | ● | ○ | ◐ | ◐ |
| [BG+17] A Systematic Approach to Implementing ABAC | ◐ | ◐ | ● | / | ◐ | ○ |
| [XP+12] Automated Extraction of Security Policies from Natural-language Software Documents | ◐ | ◐ | ○ | / | / | / |
| [XZ+23] Log2Policy: An Approach to Generate Fine-Grained Access Control Rules for Microservices from Scratch | ◐ | / | ◐ | ● | ◐ | ● |
| [LC+21] Automatic Policy Generation for Inter-Service Access Control of Microservices | ◐ | / | ◐ | ● | ◐ | ● |

**Table 3.1:** Assessment of Existing Literature Based on Requirements Catalog

As depicted in Table 3.1, none of the assessed publications fulfills all the established requirements. The research gaps are derived from the missing requirements and are established in the following.

To systematically integrate policy-driven authorization in a microservice-based application, developers should be supported throughout the development phases (R1). As presented in Table 3.1, none of the

selected publications integrates the aspect of authorization in all phases of the application development (R1). The special characteristics, such as the API specifications of microservice-based applications, have not been considered. In addition, the definition of authorization requirements has not been investigated (R2). It is unclear how the knowledge of what must be authorized should be established. As presented by the automated creation of authorization policies [XP+12; XZ+23; LC+21], the authorization policies are not as fine-grained as possible (R3). The underlying issue is the lack of understanding of what must be authorized. This marks the first research gap addressed in this thesis.

While the authors of bottom-up approaches claim that the manual creation of authorization policies is too complex and potentially costly, their solutions do not deliver fine-granular authorization (R3). In the presented bottom-up approaches [LC+21; XZ+23] which provide authorization between microservices (R4), the authorization granularity can be considered coarse-grained. Fine-grained authorization policies would only be possible if additional development artifacts are considered. The approach by Nehme et al. [NJ+18] does consider fine-grained authorization between microservices but requires additional authorization logic in the microservice implementation. Therefore, the second research gap is marked as the fine-grained authorization between microservices. This requires a systematic approach to create S2S authorization policies, which is embedded in the development process and includes development artifacts.

To support the reusability and maintainability of microservices, the authorization logic must be removed from the microservice by externalizing the authorization logic (R5). To further remove a single point of failure, the authorization mechanism can be distributed (R6). In the context of microservices, this combination has not yet been achieved. Approaches either use a centralized authorization service (e.g., [NJ+18], [SH+21]) or require the addition of authorization mechanisms in the source code as the granularity of the authorization policies is too coarse-grained (e.g., [XZ+23], [LC+21]). Thus, the third research gap is the complete externalized and decentralized authorization.

### 3.3.1 Reference to Further Chapters

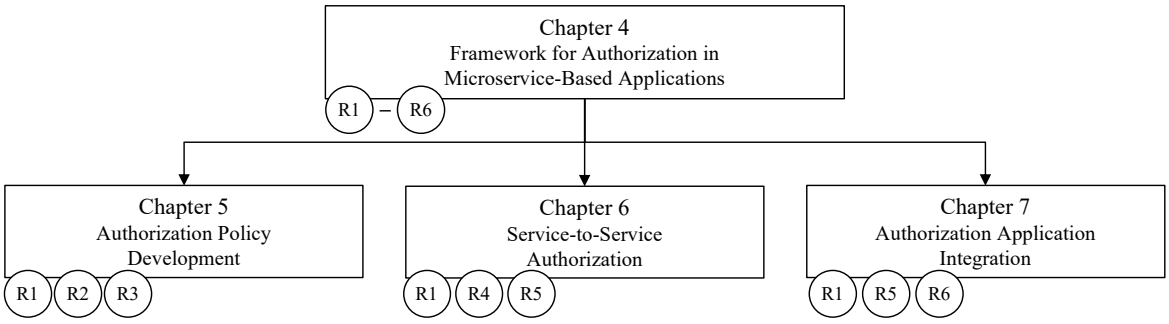


**Figure 3.8:** Classification of the Requirements in the Following Chapters

The remaining chapters of this thesis cover several requirements elaborated in this chapter. Figure 3.8 provides an overview of the contributions and the addressed requirements. In Chapter 4, an overview of the Microservice Authorization Framework (MAF) is introduced, which addresses the requirements R1 to R6. The development of fine-grained authorization policies is established in Chapter 5 which addresses the requirements R1, R2, and R3. To create the authorization policies, authorization requirements are introduced. These authorization requirements are also incorporated in the systematic creation of S2S authorization policies presented in Chapter 6, which covers the requirements R1, R4, and R5. Finally, Chapter 7 describes the enforcement of the previously created authorization policies in a microservice-based application. Authorization is performed externalized and decentralized, thus addressing the requirements R1, R5, and R6.

# 4 Framework for Authorization in Microservice-Based Applications

The development of microservice-based applications has gained increasing popularity in industry and research [SL20; BG+22; Sol-Mic]. This trend is accompanied by a growing amount of technical solutions supporting the development of microservice-based applications (e.g., containerization, Application Programming Interfaces (APIs)). The introduction of microservices can increase the overall complexity due to the inherent distributed nature of the microservice architecture [DG+17]. The secure communication among microservices or the access control on a microservice level are major challenges [AC22]. Another security issue is centralized authorization, which contradicts the loose coupling of the microservice architecture [PS+21; PS+22].

In this chapter, we introduce the comprehensive Microservice Authorization Framework (MAF), which provides a systematic approach to authorization in microservices. The framework enables developers to systematically integrate fine-grained authorization in their microservice-based applications. This requires a modification to all phases of the development, from the analysis to the deployment and operations, as well as additional steps to the development.

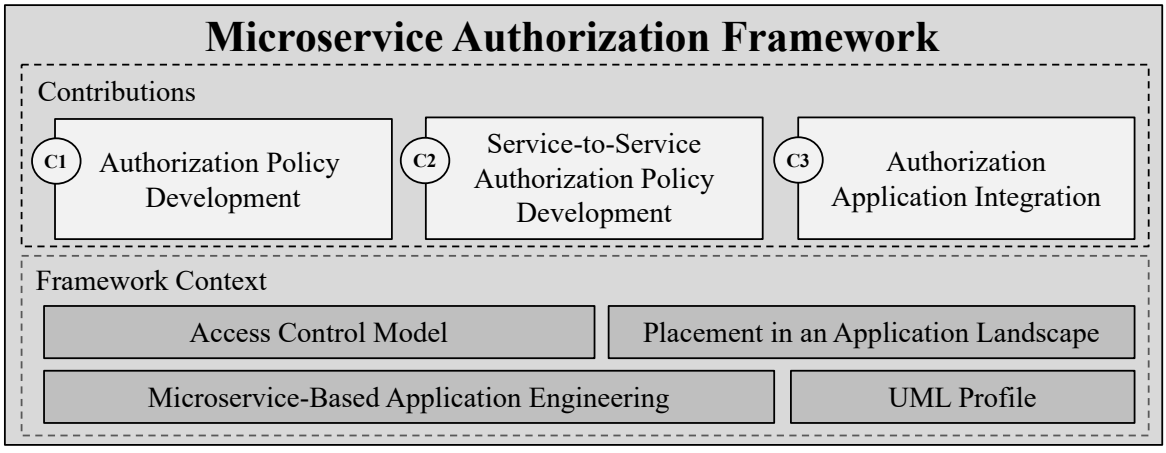


**Figure 4.1:** Overview of the Microservice Authorization Framework (MAF)

MAF is the central scientific contribution of this thesis. Figure 4.1 provides an overview of the framework. Overall, MAF contains three main contributions: First, the authorization policy development in contribution C1 provides a systematic process to create fine-grained authorization policies. The

goal of the process is to reduce the complexity of creating authorization policies. Further, the approach aims at using existing software development artifacts to derive Attribute Based Access Control (ABAC) policies. To secure the communication among microservices, the second contribution C2 addresses the Service-to-Service (S2S) authorization policy development. This requires an in-depth understanding of the communication between microservices. Therefore, additional S2S authorization policies are created. The third contribution C3 introduces the authorization application integration. The proposed integration of authorization components into the microservice architecture maintains the loose coupling among microservices.

Next to the contributions of the MAF, Section 4.2 introduces the context for the employment of the framework. This includes an overview of the access control model ABAC used in this thesis. In addition, the placement and use of the MAF in a larger application landscape is depicted. The concepts of microservice-based applications and the development thereof build the foundation of the MAF and are therefore briefly addressed. This thesis employs the Unified Modeling Language (UML) (see premises in Section 1.5) to model software artifacts [OMG-UML]. UML provides a defined set of modeling elements that can be extended through metamodels. For the modeling of authorization artifacts, an authorization UML profile is introduced. Finally, Section 4.3 provides a summary of the MAF and refers to the next chapters.

## 4.1 Contributions

Figure 4.2 depicts a more detailed view of our contributions. The basis of the framework is a systematic software engineering process following the phases of analysis, design, implementation and test, and deployment and operations. The artifacts from the systematic engineering process are used as input for the framework. Since modern software development moved from sequential development models (e.g., waterfall) to more iterative and agile methods (e.g., Scrum [AS+02]), the contributions shown in Figure 4.2 also support iterative development.

The core contributions of the framework are the derivation and implementation of authorization policies. The derivation of policies is performed from two viewpoints: First, the authorization of users interacting with a microservice-based application is introduced in contribution C1. This derivation relies on the definition of what users can do with an application. The requirements artifacts of the analysis phase define the required knowledge and use it as input for the derivation. The second viewpoint introduced in contribution C2 focuses on the authorization between microservices. The interaction between microservices can be a result of an interaction of a user with the application or a request initiated by a microservice. To derive authorization policies for the S2S communication, the artifacts from contribution C1 and the development artifacts of the microservice-based application are

used as input. An important aspect is the design of the orchestration of the application, which defines how and when the microservices interact with each other.

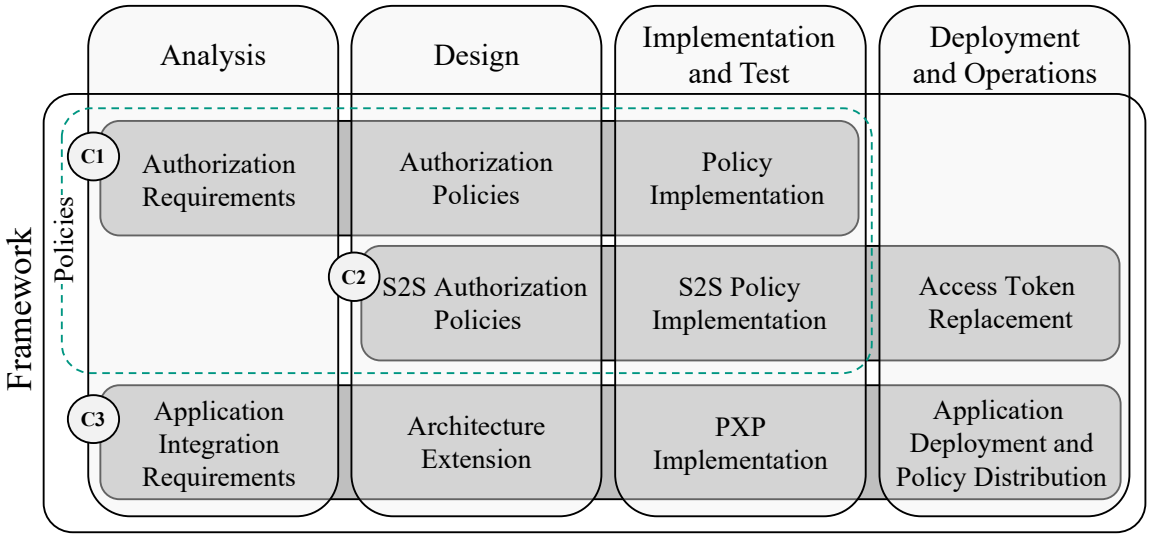


**Figure 4.2:** Contributions of the Microservice Authorization Framework

The contribution C3 focuses on the integration of authorization in the microservice-based application by enforcing the policies created in contributions C1 and C2. This requires an extension of the software architecture and the implementation of the PXPs required by ABAC (see Section 2.5.2). In addition, the systematic deployment of the architecture and the distribution of authorization policies are introduced.

### 4.1.1 C1 - Authorization Policy Development

The first contribution of MAF is a systematic process for the derivation and implementation of authorization policies. Thereby, the viewpoint of a human user interacting with a microservice-based application is taken. Figure 4.2 shows the proposed process that takes place throughout the analysis, design, as well as implementation and test phases. This requires an existing development approach for a microservice-based application, that can be extended (see Premise 3 in Section 1.5). In each phase, a dedicated authorization artifact is introduced. The foundation for the introduced authorization artifacts are the existing development artifacts (e.g., use cases or API specifications) used to develop a microservice-based application.

The analysis phase is complemented by an authorization requirement. The authorization requirement specifies what should be authorized. In the design phase, the authorization requirement is transformed into an authorization policy. The authorization policy is independent of a policy language and contains boolean statements including the attribute names resulting from the knowledge available in the

design phase. The API specification of a microservice is the underlying artifact to determine the required attributes. Subsequently, the authorization policy is implemented in a policy language. This thesis uses the policy language Rego, as it allows implementing an authorization policy declarative, similar to the source code of an application [OP-Do]. The proposed process provides traceability between authorization artifacts and the development artifacts of a microservice-based application. To ensure a systematic derivation of the authorization artifacts, for each authorization requirement created throughout the analysis phase, an authorization policy is created in the design phase, which is subsequently implemented in the implementation and test phase. The authorization artifacts from the user perspective created in C1 build the foundation for the S2S authorization introduced in C2.

### 4.1.2 C2 - Service-to-Service Authorization Policy Development

The second contribution C2 targets the fine-grained authorization between microservices. Thereby, compared to contribution C1, the viewpoint is shifted from the user interacting with the microservice-based application towards microservices inside a microservice-based application interacting with each other. While each microservice should provide its business logic through a well-defined API, microservices can require data from other microservices or perform functionality to other microservices [DG+17]. These S2S requests are typically initiated by a user interacting with the microservice-based application. Therefore, to develop the S2S authorization policies, the authorization artifacts created in C1 are used as input to determine the S2S authorization policies. Unlike contribution C1, the process to develop the S2S authorization policies spans through the phases of design, and implementation and test. The analysis phase is not considered because the S2S interaction is a result of design decisions made in the design phase. In the design phase, the S2S authorization policies are derived by first analyzing the software architecture of the microservice-based application and identifying the occurring S2S requests. Subsequently, the S2S authorization policies are derived with the help of the respective authorization requirements created in C1 that belong to the request responsible for initiating the S2S request. The S2S authorization policies are then implemented in a policy language in the implementation and test phase.

In addition to the development of S2S authorization policies, the technical modifications required by a microservice to support S2S authorization are introduced. The microservice must support identity propagation to delegate the credentials (i.e., access token) through the S2S request chain [YB18]. This allows to determine the subject responsible for the S2S request at the target microservice, which supports fine-grained authorization decisions (see Section 2.6.1). However, the distribution of credentials through a microservice-based application can lead to the leakage of said credentials by a compromised (micro)service. Therefore, we propose the component TokenHider in the deployment and operations phase, which removes the credential from the microservice and replaces it with a

temporary identifier. This allows to limit the availability of the access tokens to the authorization components and prevents the leakage or the misuse of a credential by a (compromised) microservice.

### 4.1.3 C3 - Authorization Application Integration

The last contribution C3 elaborates on the integration of policy-driven authorization into a microservice-based application. C3 enforces the authorization policies created in C1 and C2. In particular, an architectural point of view and the technical realization are considered. In the analysis phase, the requirements for the integration are elicited. This includes the access control mechanism and the architectural requirements. An important aspect is the externalization as well as the decentralization of the authorization logic (see Section 3.1). To realize the requirements from the analysis phase, an extension of the architecture of the microservice-based application is introduced in the design phase. For every microservice, we introduce dedicated Policy Enforcement Point (PEP) and Policy Decision Point (PDP) components. This allows to effectively externalize and decentralize the authorization decisions. The implementation and test phase proposes a realization of the Policy Information Point (PIP) by directly accessing the backing service of a microservice. This provides a high topicality of attributes. Finally, the automated deployment of the microservice-based application is presented, which includes the additional authorization components required for externalized and decentralized authorization. Furthermore, we propose a mechanism for the systematic distribution of the authorization policies implemented in C1 and C2. This supports the decentralized authorization by allowing the PDPs to frequently retrieve the most recent policies.

## 4.2 Framework Context

This section describes the context in which the MAF is employed. This includes a brief overview of the used access control model, the placement in a larger application landscape, and an overview of microservice-based application engineering. Furthermore, we provide a UML profile to include authorization stereotypes to model the architectural extensions in MAF.

### 4.2.1 Access Control Model

The MAF uses Attribute Based Access Control (ABAC) as an access control model (see Premise 1 in Section 1.5). ABAC allows performing fine-grained authorization decisions based on the use of attributes [HF+14]. This provides increased flexibility while avoiding problems such as role explosion (i.e., an unmanageable amount of roles), which can happen in Role Based Access Control (RBAC) [EK10]. Unfortunately, as Servos and Osborn identified in their state-of-the-art review, ABAC does

not have a uniformly accepted formal model [SO17]. Instead, various models have been proposed for specific domains (see Section 2.5). While the core idea of performing authorization decisions based on attributes remains predominately the same, the details change depending on the selected ABAC model. For instance, the terminology changes, e.g., Yuan and Tong use *resources* instead of *objects* in their ABAC model for web services [YT05]. Other examples, such as the Attribute Based Multipolicy Access Control model for grid computing by Lang et al., provide action attributes in addition to subject, object, and environmental attributes [LF+09], while the National Institute of Standards and Technology (NIST) special publication [HF+14] does explicitly not consider action attributes.

Since microservice-based applications are web applications [FL14], this thesis employs the ABAC model for web services as proposed by Yuan and Tong [YT05]. However, the terminology used by the NIST publication [HF+14] has been adopted. Section 2.5.1 further describes the resulting definition. Fundamentally, this thesis assumes that a subject performs an action to an object at a time. One or more conditions may apply and must be met for the subject to successfully perform the desired action. Authorization policies are then used to specify the conditions into boolean rules.

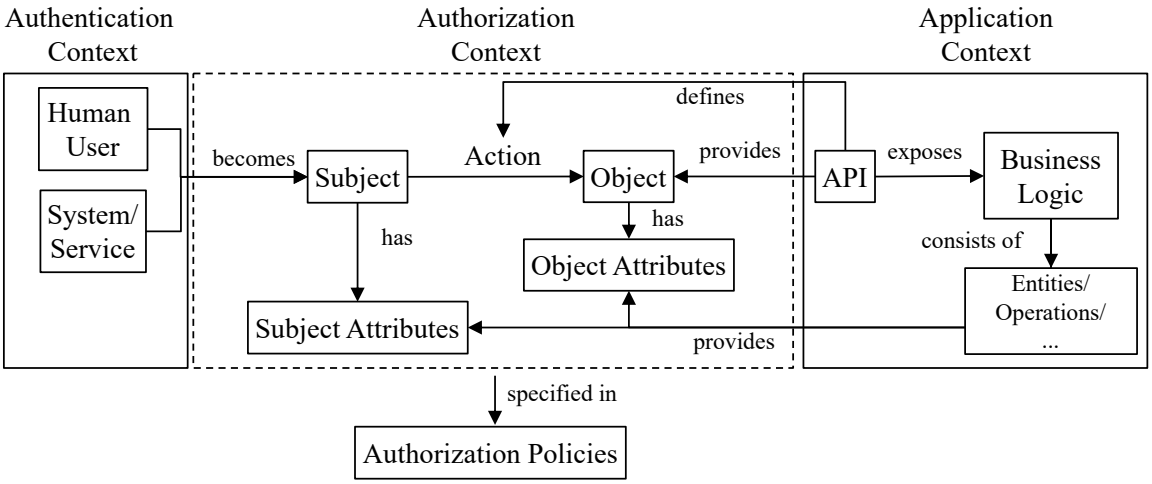


**Figure 4.3:** ABAC Terminology Used by the MAF

Figure 4.3 illustrates the ABAC terminology utilized by the MAF, which is partitioned into three contexts. The first context is that of authentication, wherein a human user or a system/ microservice seeks to gain access to the API of an application. The MAF operates under the assumption that a user has already been authenticated, e.g., through the use of a Single Sign-On (SSO) provider. Subsequently, the application context represents a microservice-based application utilizing MAF. A salient feature of these applications is that they expose their business logic through a (web) API. From an internal perspective, a microservice oversees the management of multiple entities or operations, thereby provisioning the business logic. To establish a connection between the aforementioned authentication context and the application context, the authorization context is introduced as a central

element. In the authorization context, a human user or a system is designated as a subject, attempting to perform an action on an object. The object is provided by the respective API offered by the application. Similarly, the API specifies which action can be performed. The available attributes depend upon the authentication context or application context. The authorization policies specify under which circumstances the authorization context is possible (i.e., request is allowed). For this purpose, the attributes of the subjects, objects, and the environment are documented in a policy through various conditions.

### 4.2.2 Placement in an Application Landscape

In an organizational context, an application such as the CarRentalApp is only one of many in a landscape of applications. Figure 4.4 depicts a structure of an exemplary organization. The organization has a hierarchy representing the responsibilities of various employees through different roles and groups. The position of an employee indicates which applications a user can access or what operations a user can perform. For example, an employee working in the IT department has access to a Git service while employees working in accounting do not. As shown in Figure 4.4, below the roof of the organization are the application landscape and the infrastructure services.

The foundation of the organization are infrastructure services which are used across this organization. Such services include virtualized infrastructures (e.g., Virtual Machines (VMs)) or databases. An important aspect of infrastructure services is the provision of an Identity and Access Management (IAM) system. Among others, an IAM system can provide a user storage containing the data of an organization, employees, and the necessary means to authenticate the users of the organization [IA+18]. The IAM system allows unified access throughout the various applications of the organization (e.g., through SSO).

Based on the infrastructure services, the organization provides an application landscape. In the case of the example in Figure 4.4, Application A, Application B, and the CarRentalApp exist. While Application A is a standalone application, other applications such as Application B and the CarRentalApp can rely on additional services that might be shared between applications. The applications and services run on the provided infrastructure services. It should be noted that the infrastructure services or the applications in Figure 4.4 can run either on-premise or in the cloud. Regardless of the physical location of an application or a service, each application or service integrates the authentication mechanisms provided by the organization. However, due to the specifics of an application, each application has an individual authorization.

Since access control requires authentication and authorization (see Section 2.4), the authentication and authorization of an application must be aligned with the mechanisms provided by the organization. For authentication, an organization typically has a SSO provider which can be used by the applications

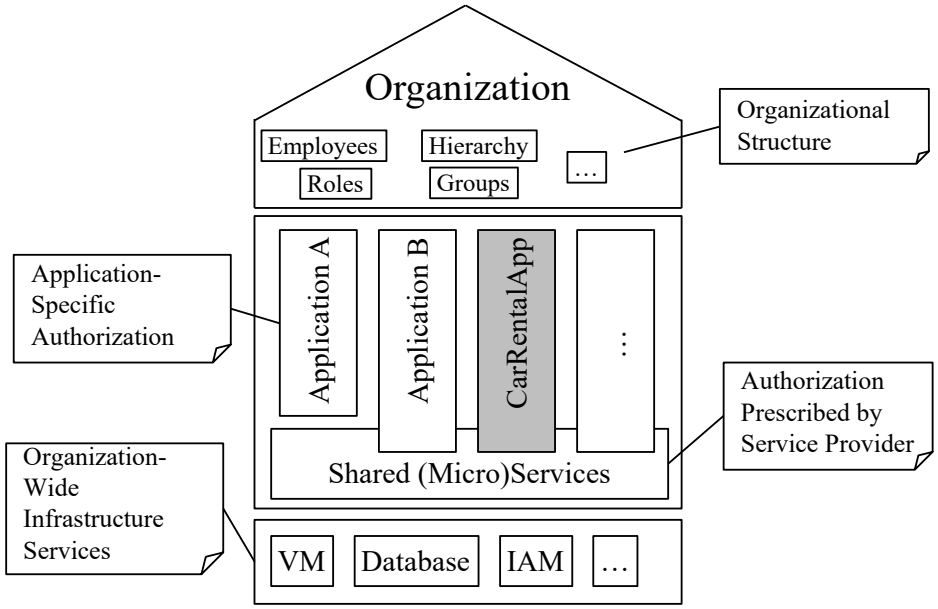


**Figure 4.4:** Applications in an Organizational Context

(e.g., using OpenID Connect (OIDC)). For the authorization, the hierarchy and the organizational structure must be mapped to the authorization of the application. For example, Application A might have an administrative role. When integrating Application A in the application landscape, the organizational role (e.g., IT administrator) must be mapped to the administrative role of Application A.

The focus of the MAF is set to supporting the development of authorization for a microservice-based application. By applying the MAF to the development of the application CarRentalApp, the authorization mechanism can be tailored to the application and the granularity of authorization can be as fine-grained as possible. However, when relying on external services that are used by or integrated into the application, the external service provider will dictate how to interact with the service. This includes how the authorization is performed (e.g., inside the external service). The MAF does therefore not consider authorization of an external service.

However, by applying MAF to the development of all applications inside an organization, a uniform authorization structure can be created. This can lead to synergies such as a uniform definition of authorization policies, the reusability of (micro)services, or a single management of authorization policies.

### 4.2.3 Microservice-Based Application Engineering

MAF considers a microservice-based application as an application consisting of multiple microservices. Thereby, microservices are used as the building blocks to provide the business logic of the microservice-based application [DG+17]. Each microservice is a (small) service providing a subset of the overall business logic. Microservices can be developed by small development teams using different programming languages [Ne15]. However, in order for the User Interface (UI) to communicate with a microservice or for microservices to communicate among another, the API must be defined. To standardize the specification of APIs, formats such as OpenAPI can be used [Ope-Spe].

The development of a microservice-based application will depend upon the environment in which it is developed. For instance, Schneider provides a systematic development approach of a microservice-based application using Domain-Driven Design (DDD) [Sc24]. The availability of development artifacts and their derivation will vary between organizations or developers creating a microservice-based application. For this reason, the MAF does not set more detailed requirements for the development of microservice-based applications. However, for the MAF to be usable in the development of a microservice-based application, two general assumptions towards the development are made.

First, the development follows the phases of analysis, design, implementation and test, and deployment and operations of a Software Development Life Cycle (SDLC). In the analysis phase, the functional requirements of the application must be collected. The design phase defines the structure of the application, which is subsequently implemented and tested in the implementation and test phase. Finally, the application is deployed and operated (e.g., monitored) in the deployment and operations phase. The use of development phases should ensure that development artifacts are a result of the previous phase (i.e., traceability).

Second, the MAF requires an API that is exposed by a microservice. As presented in Section 4.2.1, the API is required to perform authorization decisions. Ideally, the API is specified using an explicit specification language such as OpenAPI [Ope-Spe] or Protocol Buffers [Go-Pro].

### 4.2.4 UML Profile

MAF uses UML to model software engineering artifacts. UML provides various types of diagrams, which can be categorized into static and dynamic diagrams [OMG-UML]. The modeling of software architecture has different views, such as the logical view and physical view. The physical architecture can be modeled using the UML deployment diagram, which allows modeling physical compute nodes running executable artifacts. The logical view can be modeled using the UML component diagram. The component diagram allows structuring software components and putting them into relation by defining interfaces using the ball and socket symbol.

By default, UML does not include mechanisms for security. However, the default UML specification can be extended to fit the respective use case using different approaches. UML uses a four-layered architecture separating different conceptual levels [OMG-UML]. The UML specification can be extended by either creating a new metamodel or creating a UML profile for a metamodel in the M2 layer [FV04]. Lodderstedt et al. [LB+02] introduce SecureUML which allows modeling users, roles, and permissions on UML elements. Further, authorization constraints on UML elements can be described using the Object Constraint Language (OCL) constraints. Dobmeier and Pernul [DP06] introduce a metamodel to describe ABAC policies. Both extensions use a metamodel to extend UML.

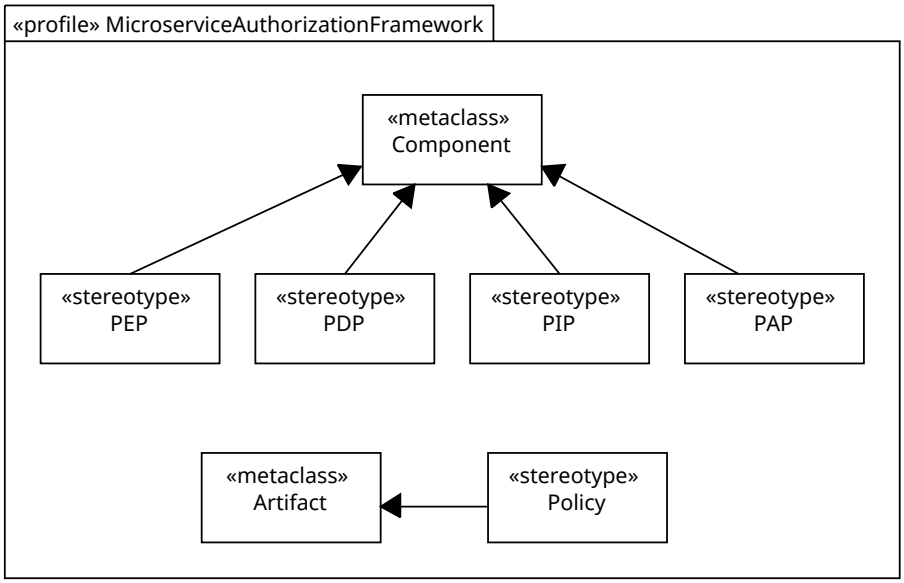


**Figure 4.5:** UML Profile

Compared to the above-mentioned metamodels, we provide a UML profile which is modeled as a package diagram with the stereotype «profile» [FV04]. Figure 4.5 displays the UML profile for the MAF. The goal for the UML profile is to provide a uniform modeling of authorization components throughout this thesis. The «metaclass» Component is extended by stereotypes for the PEP, PDP, PIP, and Policy Administration Point (PAP) that are required by the eXtensible Access Control Markup Language (XACML) reference architecture (see Section 2.5). For the extension, the UML inheritance is used. These stereotypes allow defining logical components when modeling a software architecture. Furthermore, an extension for the «metaclass» Artifact is created by introducing the «stereotype» Policy. The additional stereotypes are used when describing UML artifacts in MAF.

## 4.3 Summary

This chapter provides an overview of the Microservice Authorization Framework and its context. The framework contains the three primary contributions of this thesis and sets the contributions in relation to each other. Figure 4.6 provides an overview of the framework including the authorization policy development (C1), the development of S2S authorization policies (C2), and the authorization application integration (C3). Furthermore, the framework context is introduced. This includes the definition of the terms used in this thesis and the classification of the authorization used in an organizational context. A UML profile is introduced for modeling authorization components in UML used throughout this thesis. Each of the subsequent Chapters 5 to 7 introduces a contribution in detail. Subsequently, Chapter 8 provides a validation for each of the contributions.

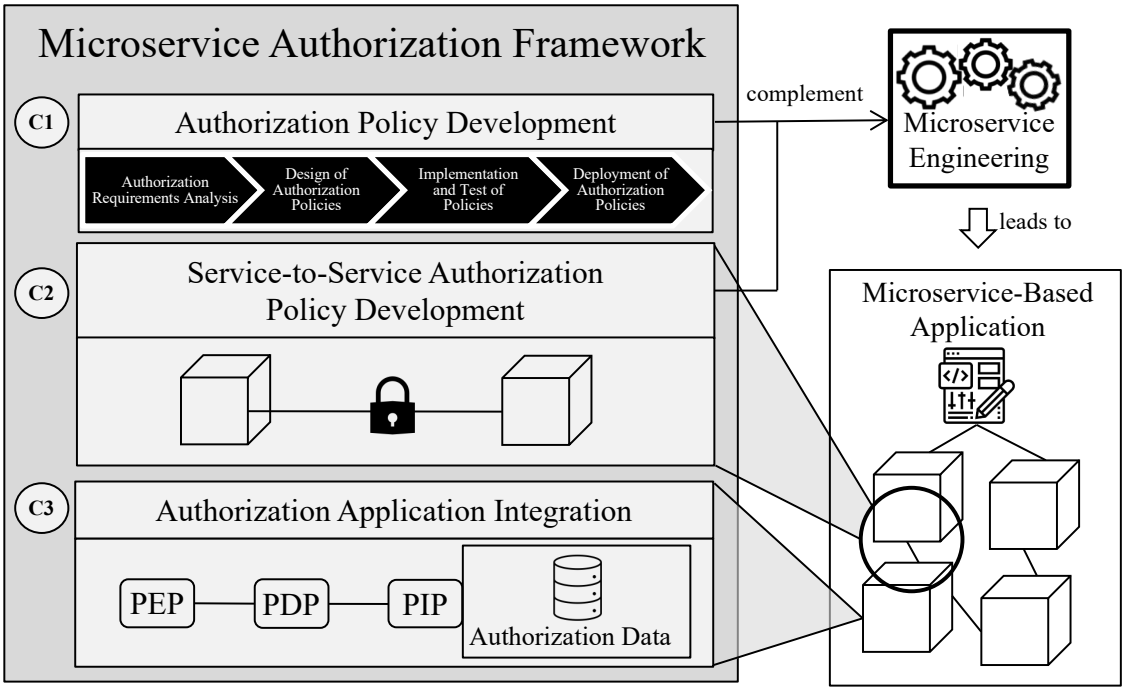


**Figure 4.6:** Overview of the Microservice Authorization Framework

# 5 Authorization Policy Development

This chapter addresses the systematic development of authorization policies, also known as policy engineering [DM+18]. There are several approaches to the definition of a policy. Das et al. classify policy engineering into top-down, bottom-up, and general approaches [DM+18]. The top-down approach identifies policies based on natural language documents. This also requires the processing of natural language documents. Bottom-up approaches create policies based on the investigation of existing data that is created during the execution of a system (e.g., logs). Approaches classified as general are based on, e.g., risk definition. To address the requirement *R1 - Embedding Authorization Into Development* (see Section 3.1), a top-down approach is pursued in this section. However, as Das et al. note, the definition of a top-down process based on natural language documents is inexact [DM+18]. This is due to the nature of natural language documents, which depends on the structure and the content of the person defining the document. Wahsheh et al. introduce a policy life-cycle for policy engineering which includes the policy requirements analysis, policy design, policy implementation, and policy enforcement [WA08]. They regard policy engineering as an additional part next to software engineering.

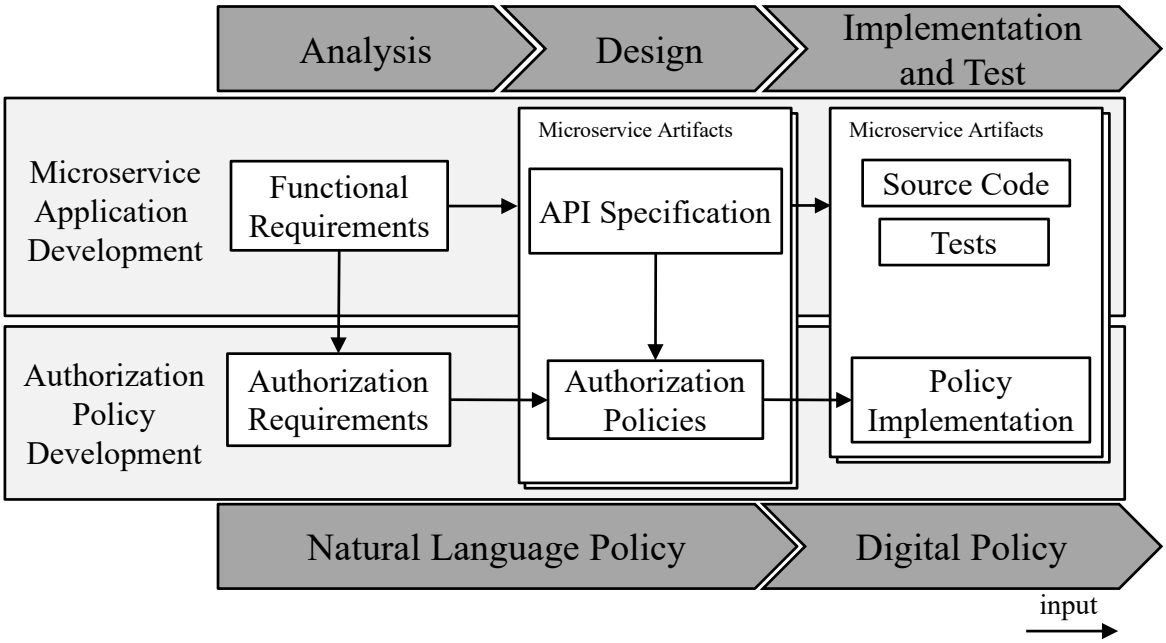


**Figure 5.1:** Microservice-Based Application Development with Authorization Policy Development

Figure 5.1 shows an overview of the proposed authorization policy development, which is the contribution of this chapter. This assumes a development following the phases of the Software Development Life Cycle (SDLC) (see Section 2.1), namely the analysis phase, design phase, and implementation and test phase, which are depicted in the upper half of Figure 5.1. To create a microservice-based application, the analysis artifacts (e.g., functional requirements) are transformed into design artifacts. The special characteristic of microservice-based applications is the division of the business logic into a set of microservices. Each microservice receives its own design artifacts. The inherent distributed nature of microservices requires a well-defined API specification which is aligned with the business logic provided by the microservice [DG+17]. Therefore, the API specification is the crucial design artifact of a microservice. Following the design phase, each microservice is implemented and tested. It is possible for a microservice to be developed by different teams in different programming languages [Ne15].

The authorization policy development is presented in the lower half of Figure 5.1. It considers explicit authorization artifacts and process steps in each development phase to take authorization into account. The form of an authorization policy is contingent upon the information available at a given point in the development process. According to Hu et al. [HF+14], Natural Language Policy (NLP) and Digital Policy (DP) should be considered. NLPs are statements governing the management and access of objects that are human-readable, which can subsequently be transformed into machine-enforceable access control policies. DPs are access control rules that can be compiled into machine executable code. Subject, object, attributes, and rules are the building blocks for a digital policy [HF+14]. Throughout the development phases of a microservice-based application, the available information regarding the authorization changes in clarity and structure. The authorization artifacts are derived based on development artifacts and, if available, previous authorization artifacts. In the analysis phase, authorization requirements are introduced which capture the conditions that must be fulfilled to provide a functional requirement (e.g., use case). In the design phase, authorization requirements are refined into an authorization policy. With the available design knowledge, the authorization policies include the necessary attributes and conditions. To provide a traceability between authorization artifacts, each authorization requirement should result in one authorization policy. Finally, in the implementation and test phase, each authorization policy results in a policy implementation. Analogous to the implementation of microservices, various policy languages can be used.

The remainder of this chapter is structured along the development phases required for the introduction of the authorization policy development. The focus is the introduction of the process required for the creation of the authorization artifacts. Figure 5.2 shows a high-level overview of the involved steps. The first step is the derivation of authorization requirements. Using functional requirements as input, this step is performed in the analysis phase introduced in Section 5.1. As a result, the authorization requirements are created. The authorization requirements and the API specifications

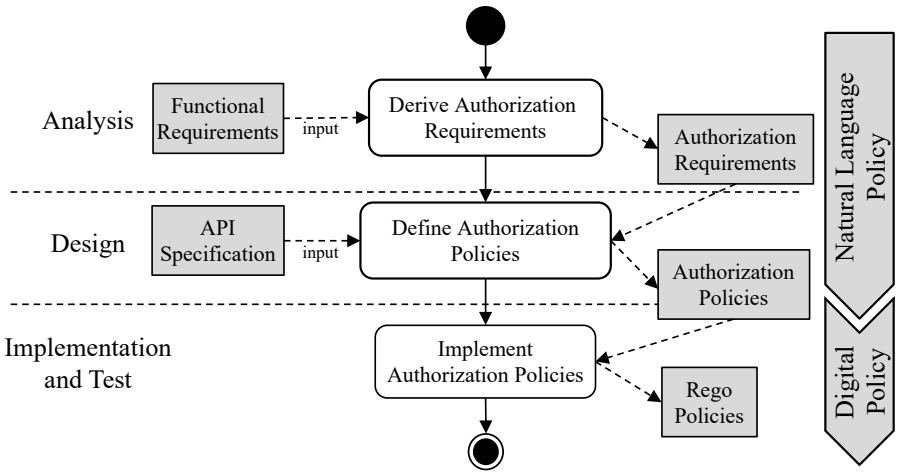


**Figure 5.2:** Process to Define Authorization Artifacts

are used in the definition of authorization policies in Section 5.2. This results in the definition of authorization policies, which are then implemented in a policy language in the implementation and test phase described in Section 5.3. In this chapter, we use the policy language Rego provided by the Open Policy Agent (OPA) as an example policy language. The development of the authorization artifacts is introduced using the CarRentalApp (see Section 1.4.4).

## 5.1 Analysis

The analysis phase elicits the functional requirements of the application. To already consider authorization during the analysis phase, the authorization requirement is introduced as a dedicated artifact [Fi03]. The authorization requirements aim at providing an initial structure for the realization of an ABAC authorization policy. The foundation for the authorization requirements are the functional requirements specifying how a user interacts with the system (e.g., use cases). Listing 5.1 presents an authorization requirement for the use case *List Customer Rentals*, which contains the necessary information required by ABAC such as the subject (*customer*), action (*list*), object (*rentals*), and further conditions that use attributes (e.g., *RentalBelongsToCustomer*).

```
1 ---CustomerListCustomerRentals---
2 subject customer
3 can perform action list
4 on object rentals if
5 RentalBelongsToCustomer
```

**Listing 5.1:** Exemplary Authorization Requirements for CarRentalApp

An authorization requirement is the result of the process presented in Figure 5.3. To specify an authorization requirement, the functional requirements are used as input to create authorization requirements. Initially, the involved subject, objects, and actions must be identified. These are the core building blocks of ABAC (see Section 2.5). Next, the conditions which must be met for a subject to perform an action to an object are identified. Finally, with the identified building blocks of an ABAC policy, the authorization requirement can be formulated.

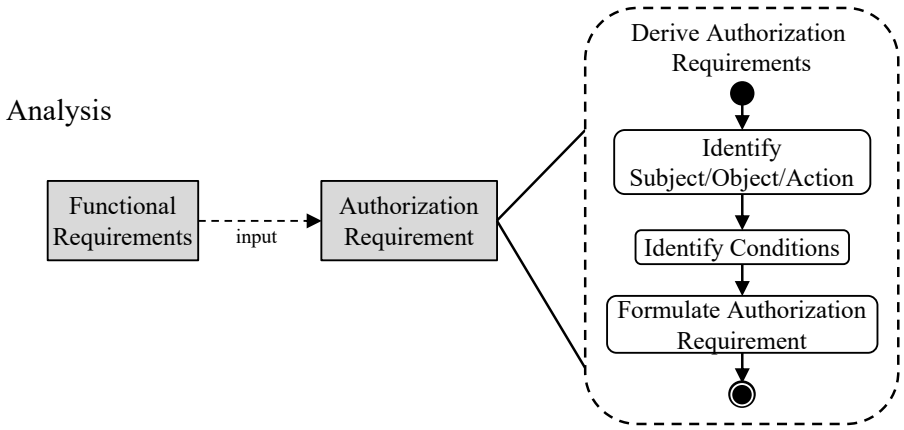


**Figure 5.3:** Process to Derive Authorization Requirements

As introduced in Section 4.2.1, Microservice Authorization Framework (MAF) relies on the Attribute Based Access Control (ABAC) model introduced by Yuan and Tong [YT05]. This implies, that only one subject can perform one action on one object at a time. This restriction limits the ABAC complexity in the domain of web applications. However, this also has implications on the creation of the authorization artifacts. The foundation for the authorization requirements are the functional requirements. Ideally, these requirements reflect the limitations introduced by the ABAC model. The subsequent sections will describe the ideal case with use cases as functional requirements. Each use case will describe a goal for a single actor who wants to perform an action on one object. Since this will not always be the case, Section 5.1.4 presents ways of dealing with diverging functional requirements.

### 5.1.1 Identify Subject, Object, and Action

A common analysis artifact to specify the functional requirements are use cases, which are used to specify the functionality of CarRentalApp. Use cases are a description of interactions between an actor and a system related to a particular goal [Co00] (see Section 2.1). The goal of a use case can be defined on different levels. To derive authorization requirements, user-goal level use cases can be used. Listing 5.2 depicts a use case description from the CarRentalApp. The use case is specified using a template as suggested by Cockburn [Co00]. The content of the use case description contains a title

```
1 Title: List Rentals
2 Primary Actors: Customer
3 Secondary Actors: None
4 Preconditions: Rentals belong to Customer
5 Postconditions: None
6
7 Flow:
8 1. Customer lists his rentals.
9 2. System presents the customer with all rentals...
```

**Listing 5.2:** Use Case "List Rentals"

(line 1), the primary and secondary actors (lines 2 and 3), preconditions (line 4) and postconditions (line 5), and flows (lines 7 to 9). The use case description allows specifying the goal a user wants to achieve, how to achieve said goal, and which conditions must be met. Use cases are therefore a good source of information for setting up authorization requirements. However, this process aims to make it possible to employ other types of requirements. The prerequisite for this is the presence of the subject, the action, the object, and the conditions required for ABAC in the requirements description.

The identification of the subject, object, and action can be done by analyzing the use case description. The title of the use case is structured as proposed by Cockburn using <*verb*> <*direct object*> [Co00]. The verb reflects the action of the use case. In the example presented in Listing 5.2, the action is *List*. Hence, *Rentals* reflects the object. The subject can be identified based on the primary actors. In this case, the subject is a *customer*.

Unfortunately, not all use case descriptions allow deriving the authorization information as simple as in Listing 5.2. The use case description can also contain ambiguous actions or objects. For example, the use case list car rentals depicted in Listing 5.3 allows a fleet manager to view the rentals belonging to a car in their fleet. The subject and the action can be directly extracted from the description. The subject is a *fleet manager* and the action is *list*. However, the object is not apparent. The title suggests *Car Rentals*, while the flow in line 9 suggests that a fleet manager wants to see *rentals* for a specific car. This example highlights that the knowledge towards authorization is not clear throughout the analysis phase. Throughout the design of a microservice-based application, the ambiguities should be resolved. For example, it should be clear whether the object is called *Car Rentals* or *Rentals*.

### 5.1.2 Identify Conditions

The next step is the identification of conditions. The conditions allow creating fine-grained authorization decisions. If the use cases and use case descriptions are employed, the preconditions can be utilized to identify authorization conditions. For the use case presented in Listing 5.2, line 4 presents

```
1 Title: List Car Rentals
2 Primary Actors: Fleet manager
3 Secondary Actors: None
4 Preconditions:
5     - Rental exists
6     - The car exists in fleet of fleet manager
7 Postconditions: None
8 Flow:
9 1. Fleet manager selects to view all rentals of the car.
10 2. System presents all rentals ...
```

**Listing 5.3:** Use Case "List Car Rentals"

the condition that only the rentals that belong to a customer are presented. This condition indicates that there is a relationship (i.e., belongs), between the object *rentals* and the *subject* customer. This connection will be considered further in the design phase. When looking at the conditions of a use case description, we identify two other types of conditions.

First, conditions that are part of the business logic. For instance, in Listing 5.3, the precondition in line 5 requires that the rental exists. While this is a relevant condition for the use case, it is not necessarily relevant for the authorization. Therefore, when creating authorization requirements, it should be considered whether a condition is business logic or not. Conditions related to business logic will introduce additional complexity in the authorization policies. However, missing a relevant condition might lead to a lack of authorization granularity.

Second, it is also possible to include more complex conditions that, e.g., rely on attributes from additional objects. For example, in line 6 of Listing 5.3, the use case describes a fleet manager who is only allowed to see the rentals of a car, if the car is in the fleet manager's fleet. This introduces the object *fleet* and a relationship between the object *car* and the object *fleet* which must be evaluated. This will become relevant in the design phase, where different objects can be managed by different microservices, thus requiring an exchange of attributes between microservices.

### 5.1.3 Formulate Authorization Requirements

We propose a common structure for authorization requirements that should be employed among developers. In the analysis phase, with the help of a template, the authorization requirements can be written in natural language. Depending on the development environment, the template can be adapted to the specific needs. In addition, there should be a clear structure to name authorization requirements. Each authorization requirement should receive a unique name (or identifier) to be individually addressable. This name is used for the further development of the authorization policies.

Listing 5.4 presents an example template for authorization requirements. Each authorization requirement starts with an identifier (line 1) consistent of the name of the actor and the use case that belongs to the authorization requirement. This should be unique within the microservice-based application. Subsequently, the identified subject (line 2), action (line 3), and the object (line 4) can be entered into the template. Finally, the conditions (lines 5 and 6) are added. In this template, each condition is written in CamelCase.

```
1 ---ActorUseCaseTitle---
2 subject <subject> is allowed to perform
3 action <action> on
4 object <object> if
5 condition <Condition1InCamelCase> and/or
6 condition <Condition2InCamelCase> and/or ...
```

**Listing 5.4:** Authorization Requirement Template

Finally, Listing 5.5 presents the authorization requirement for the use case *List Car Rentals*. The unique title *FleetManagerListCarRentals* is defined in line 1. The object *car rentals* has been selected (line 4). Finally, the conditions are formulated in CamelCase in line 5.

```
1 ---FleetManagerListCarRentals---
2 subject fleet manager
3 action list on
4 object car rentals if
5 CarInFleetOfFleetManager
```

**Listing 5.5:** Exemplary Authorization Requirement "List Car Rentals"

### 5.1.4 Further Derivation Options

So far, the derivation of authorization requirements is based on use cases that reflect an ideal case. As presented in Section 2.5.1, this thesis considers an ABAC model in which one subject can perform an action on one object under a given set of conditions. This is not always the case when creating use cases or other functional requirements. For instance, a use case might have multiple primary actors or consider multiple actions or objects. We identify at least two options to address this issue. First, create multiple authorization requirements from one use case. Second, create an authorization requirement with multiple subjects/actions/objects and resolve the issues throughout the design phase, when the requirement has been transferred into the software design. However, this might also be an indicator, that the functional requirement (e.g., use case) should be refactored.

If multiple authorization requirements are to be created from one functional requirement, we propose the segmentation of the functional requirement into multiple authorization requirements following the restriction introduced by the employed ABAC model of one subject, one action, and one object. In the case of a use case with multiple subjects/actions/objects, this involves taking a closer look into the use case flow. If multiple actors are involved in a use case, an authorization requirement can be created for each actor. By analyzing the use case flow, it can be determined if the actors perform the same interactions with the system or not. If they perform the same actions on an object, the authorization requirements should only differ by the respective subject. If they perform different actions or involve different objects, the authorization requirements should include the respective action and object. Similarly, if the use case contains multiple actions, the flow must be analyzed to identify the interactions between the actor and the system. For each action with the system, an authorization requirement should be created. This provides fine-granularity due to the focus on atomic actions performed by the actor to the system. Finally, if there are multiple objects involved in the use case, these objects can be regarded as a set of objects that are accessed in a single atomic operation or multiple objects that are accessed through multiple actions. This again can also be identified from the use case flow. For each action on an object, an authorization requirement should be created.

If only one authorization requirement is created, the authorization requirement must be resolved to multiple authorization policies in the design phase. In the microservice-based application, the business logic is provided through an API. Therefore, the authorization requirement containing multiple subjects/actions/objects will likely result in multiple API requests, potentially to different API endpoints. Each API request should be authorized, thus requiring an authorization policy.

## 5.2 Design

In the design phase of a SDLC, the artifacts from the analysis phase are transformed into design artifacts. In the context of microservice-based applications, the business logic defined in the analysis phase is spread to a set of microservices. Each microservice provides its business logic through an API. The goal of this section is the systematic derivation of an authorization policy. For each authorization requirement created in the analysis phase, an authorization policy is created. This allows to maintain traceability throughout the development phases. The resulting authorization policy is similar to an authorization requirement. However, the authorization policies are assigned to the microservice that provides the respective business logic. In addition, the authorization policies use the attributes available in the design phase. Since the authorization is to be performed externalized from the microservice, the primary resource to derive these attributes is the API specification. If further attributes are required, they can be retrieved from the respective Policy Information Point (PIP). This is especially needed when using attributes from objects managed by other microservices.

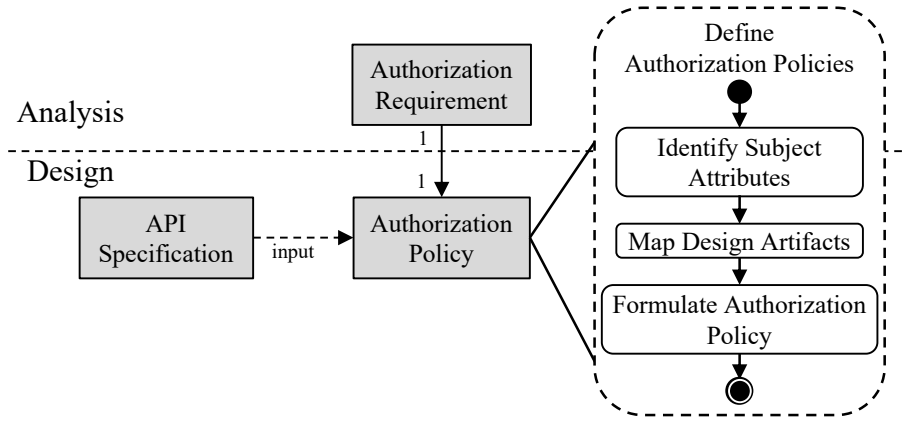


**Figure 5.4:** Process to Define Authorization Policies

Figure 5.4 presents an overview of the process to create authorization policies. The first step is the identification of the subject attributes. To perform fine-grained authorization decisions, the information about the subject using the microservice-based application must be available. This can be done, e.g., through credentials such as a JSON Web Token (JWT). An important point here is the identification of the subject. In the next step, for each authorization requirement, the respective design artifacts are mapped. This results in the identification and location of necessary objects and attributes. Finally, the authorization policy is formulated.

```
1 ---FleetManagerListCarRentals---
2 A subject can perform
3 action ListCarRentals on
4 object RentalsCollectionService if
5 Fleet.cars contains input.vin and
6 Fleet.fleetManager == subject.sub and
7 fleetmanager in subject.roles
```

**Listing 5.6:** Exemplary Authorization Policy "List Car Rentals"

An example authorization policy for the use case *List Car Rentals* (see Listing 5.5) is presented in Listing 5.6. The structure is analogous to the authorization requirement. This includes the identifier of the authorization policy (line 1). Since the use case is provided by the microservice RentalManagement, the action (line 3) and the object (line 4) are derived from the gRPC API specification of the microservice RentalManagement. In lines 5 and 6 the condition *CarInFleetOfFleetManager* is evaluated. This requires access to an object *Fleet* that is provided by the microservice FleetManagement. Finally, line 7 evaluates if the subject is a *fleet manager* by evaluating the respective role.

### 5.2.1 Identify Subject Attributes

The attributes of the subject are essential to understand who is performing a request. The process of authentication verifies the identity of a user. Authentication is a necessary step before authorization can be performed [SS94]. Typically, the identity of a subject is managed in an Identity and Access Management (IAM) system. Popular protocols to authenticate a subject are OpenID Connect (OIDC) and Security Assertion Markup Language (SAML) (see Section 2.3). If a subject is successfully authenticated, these protocols provide one or multiple objects containing attributes of a subject. In addition, these objects are typically signed, which makes it possible to verify that the object has not been tampered with.

Listing 5.7 provides an example access token of a customer. The access token is the result of a successful authentication using OIDC. The token is in the JWT format and contains the claims *sub* (line 2) and *roles* (line 4), among others. The content of the access tokens can be configured in the IAM system issuing the access token. For the authorization policies, a unique subject identifier is required. For example, the *customerId* of a *rental* object should be equal to the identifier of the access token. If OIDC is used, the claim *sub* represents the subject identifier and is required by the OIDC specification [SB+14]. Thus, if the design decision to use OIDC is made, the claim *sub* can by default be used as a subject identifier. SAML provides a similar identifier, which is not further elaborated in this section.

```
1 {
2     "sub": "alice.smith@gmail.com",
3     ...,
4     "roles": [
5       "customer"
6     ]
7 }
```

**Listing 5.7:** Exemplary Access Token for Customer

The IAM system can contain more subject attributes. For example, the age or the address can be part of the stored subject. Thus, if additional subject attributes from the IAM system are required, the system must be configured to provide the attributes through the token. If the attributes are not part of the access token but necessary for the evaluation of a condition, the attributes must be accessed through a PIP.

For the running example CarRentalApp, the attribute role is used to differentiate between a *customer* and a *fleet manager*. In the IAM system, the roles must be assigned to the respective subject. This allows to perform coarse-grained authorization decisions in the absence of further attributes. For instance, a subject with the role *fleet manager* should not be able to rent a car, while a *customer* should

not be able to list fleets. Including a role as a subject attribute represents the actors defined in the analysis phase (e.g., use case diagram) without creating an unmanageable amount of roles in the IAM system.

### 5.2.2 Map Design Artifacts

After the subject attributes have been identified, the remaining attributes that are part of an authorization requirement must be resolved. With the use of externalized authorization, the API specification of a microservice depicts the primary source of information to identify the relevant attributes. In an environment where multiple teams develop a microservice-based application, only the API specifications are likely to be known to the respective teams. Additional attributes that are not exposed through the API will be unknown to other development teams. Therefore, we will focus on the attributes that are available in the API specifications of a microservice-based application.

The goal of this process step is to find the counterpart terms from the authorization requirement in the API specification. This involves two steps: First, identify the relevant terms in the authorization requirement. Second, map the respective term to an API specification. Depending on the design of the API specification, the names will be more or less similar to the authorization requirement. Since there is a wide variety of API types, this process is demonstrated by mapping gRPC APIs and RESTful APIs (see Chapter 2).

**gRPC API** gRPC is a Remote Procedure Call (RPC) framework which can be implemented with multiple programming languages. Protocol buffers (protobuf) is the language used to specify the API. Listing 5.8 provides an excerpt of the gRPC specification for the microservice RentalManagement. The RPCs are collected in a gRPC service. Each RPC requires an input and an output. In gRPC, these are specified as messages. For example, the RPC ListCarRentals in line 3 requires a message ListCarRentalsRequest which contains an identifier called Vehicle Identification Number (VIN). The return message contains the requested rental object(s) as presented in line 12.

There is no general guideline towards structuring a gRPC API. In the gRPC specification presented in Listing 5.8, the RPCs are structured around the message *Rental* (lines 14 to 18) in the service *RentalsCollectionService*. Therefore, the object *car rentals* from the authorization requirement presented in Listing 5.5, can be mapped to the *RentalsCollectionService*. The action performed on the object can be mapped to the respective RPC of the gRPC service *RentalsCollectionService*. The response messages from the RPC contain the requested objects. The attributes can be derived from the messages as defined in the gRPC specification. For example, the Rental message contains a *customerId* (line 16) which can be used to determine if a *rental* object belongs to a customer. Analogous, the

```
1 service RentalsCollectionService {
2     rpc ListCustomerRentals(ListCustomerRentalsRequest) returns (
          ListCustomerRentalsResponse) {}
3     rpc ListCarRentals(ListCarRentalsRequest) returns (
          ListCarRentalsResponse) {}
4 }
5 message ListCarRentalsRequest {
6     string vin = 1;
7 }
8 message ListCustomerRentalsRequest {
9     string customerId = 1;
10 }
11 message ListCarRentalsResponse {
12     repeated Rental rentals = 1;
13 }
14 message Rental {
15     string id = 1;
16     string customerId = 6;
17     ..
18 }
```

**Listing 5.8:** Exemplary gRPC Specification for RentalManagement

attribute *vin* inside the request message can be used for the evaluation of the condition *CarInFleetOfFleetManager*. However, since the API specification of the microservice RentalManagement does not contain a message identifying the *Fleet*, the required information must be retrieved from another API specification. The fleet is managed by the microservice FleetManagement, which also specifies the required attributes inside its API specification. To check this condition, the attributes must therefore be retrieved from the FleetManagement microservice (i.e., through an PIP).

**RESTful API** RESTful APIs are a widely used API paradigm. A popular standard to formulate API specifications is OpenAPI provided by the OpenAPI initiative [Ope-Spe]. OpenAPI uses YAML Ain't Markup Language (YAML) or JavaScript Object Notation (JSON) to specify RESTful API specifications (see Section 2.2.1). Listing 5.9 presents an excerpt of the OpenAPI specification for the microservice RentalManagement which provides rental objects for a given car identified by a VIN. The path in line 2 defines the HTTP path to the requested object. The parameter *vin* required to access the rentals is defined in lines 4 to 6. The response of the request is defined through a reusable component called *Rental* (line 14) which is defined through a schema (line 17).

The object from the authorization requirement is mapped to the HTTP path of the API specification. The action must be translated to the respective HTTP operation that is defined in the OpenAPI specification. For example, the action *list* is translated to the HTTP operation GET while the action *add* is translated to the HTTP operation POST or PUT. The attributes are retrieved from the defined

```
1  paths:
2    /rentals/{vin}:
3      get:
4        parameters:
5          - name: vin
6            in: path
7            ..
8        responses:
9            '200':
10           content:
11             application/json:
12               schema:
13                 items:
14                   $ref: '#/components/schemas/Rental'
15 components:
16   schemas:
17     Rental:
18       type: object
19       properties:
20         id:
21           type: string
22         vin:
23           type: string
24         customerId:
25             type: string
```

**Listing 5.9:** Exemplary OpenAPI Specification for RentalManagement

schemas. The Rental object contains the attributes *id* (line 20), *vin* (line 22), and a *customerId* (line 24). Similar to the previous gRPC example, the fleet defined in microservice FleetManagement is required for the evaluation of the authorization requirements. Therefore, attributes must be retrieved from FleetManagement.

### 5.2.3 Formulate Authorization Policy

The final step is the formulation of the authorization policy. Listing 5.10 presents an example authorization policy for the use case *List Car Rentals*. The structure has been adapted from the authorization requirement. The authorization policy consists of five logical statements. Each statement must be evaluated as true in order for the authorization policy to be fulfilled. The input used in the authorization policy depicts the request that is performed by the subject. The content of the request (e.g., parameters, HTTP headers) is forwarded from the PEP to the PDP. Depending on the selected API paradigm and the PDP, the structure of the input can vary. This is further elaborated in the next section. To compare attributes and values received from the input, a consistent set of comparison operators should be used. In the example presented in Listing 5.10, the operators *in* and *contains*

are used to check if a key is in a set of values. The operator == is used to compare two values for equality. All comparisons should result in true/false values. In line 2, the subject must have the role *fleetmanager*. The action and the object must be *ListCarRentals* (line 3) and *RentalsCollectionService* (line 4) respectively. To evaluate the condition *CarInFleetOfFleetManager*, an object *Fleet* must exist that contains a car with a requested *vin* (line 5) and has a fleet manager with the identifier *sub* of the subject (line 6).

```
1 ---FleetManagerListCarRentals---
2 fleetmanager in input.subject.roles
3 "ListCarRentals" == input.action
4 "RentalsCollectionService" == input.object
5 Fleet.cars contains input.vin
6 Fleet.fleetManager == input.subject.sub
```

**Listing 5.10:** Exemplary Authorization Policy "List Car Rentals"

Creating the authorization policies based on the authorization requirements and the design artifacts of the microservices, specifically the API specification, allows them to be independent of a specific policy language (e.g., XACML or Rego). This provides two advantages to the development of authorization policies: First, similar to the different programming languages that might be used by development teams for a microservice, these development teams can select different policy languages. Second, language-agnostic authorization policies allow exchanging the policy language dependent on the use of the microservice-based application. For instance, this allows the same microservice-based application to be used in multiple environments (e.g., two companies) requiring different policy languages (e.g., due to corporate wide requirements).

## 5.3 Implementation and Test

Throughout the analysis phase and design phase, the authorization requirements and authorization policies have been classified as natural language policy (see Figure 5.1). In the implementation phase, the defined authorization policies are transferred into digital policies, i.e., implemented in a policy language. In our MAF, the policy language Rego is used. Rego is the policy language provided by OPA (see Section 2.5.4) [OP-Do]. The policy language is declarative and enables the creation of authorization policies as code, which is more comprehensible when compared to XML (as utilized by XACML). OPA is developed for distributed applications and is also used in related work addressing authorization such as ThunQ introduced by Sauwens et al. [SH+21]. Since the authorization policies can also be implemented in other policy languages, Appendix A.3 provides an example in the policy language Casbin.

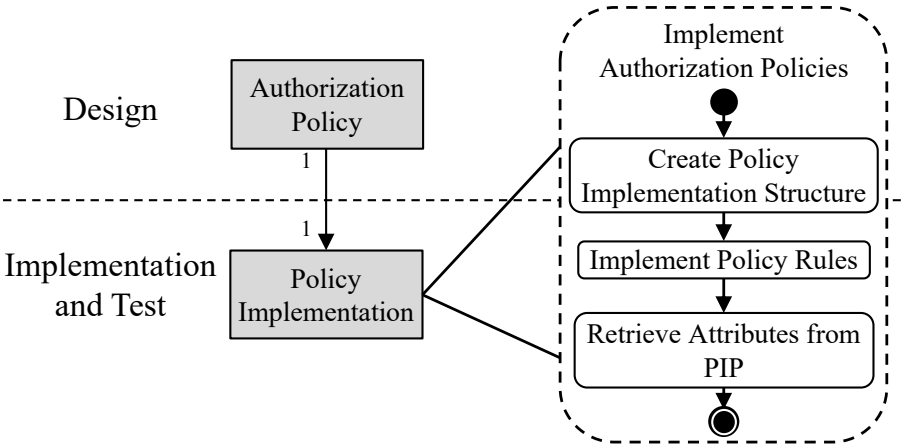


**Figure 5.5:** Process to Implement Authorization Policies

Figure 5.5 depicts an overview of the process to implement authorization policies. For each authorization policy created in the design phase, a Rego authorization policy is implemented. The process applies software implementation practices to the implementation of the authorization policies (i.e., to reduce code duplication). The first step is the creation of a policy implementation structure. For each authorization policy, a dedicated Rego file is created. Subsequently, each rule of the authorization policy is implemented. Finally, the logic to retrieve the attributes from the PIP is implemented. In addition, Section 5.3.4 provides a brief example of unit tests for authorization policies created with Rego.

```
1 #---FleetManagerListCarRentals---
2 allow {
3   "fleetmanager" in input.subject.roles
4   "ListCarRentals" == input.parsed_path[1]
5   "rentalmanagement.RentalsCollectionService" == input.parsed_path[0]
6   true == data.car_in_fleet_of_fleetmanager(input.parsed_body.vin, input.
        subject.sub)
7 }
```

**Listing 5.11:** Example Authorization Policy Implementation in Rego

Listing 5.11 presents an exemplary Rego policy implementation for the use case *List Car Rentals* (see Listing 5.2). In Rego, each authorization policy is implemented inside a so-called allow statement, which is denoted by curly brackets (lines 2 and 7). Each allow statement contains a set of rules. A rule is a statement that can be evaluated to true or false. If all rules evaluate to true, the allow statement will become true. Subsequently, if one rule evaluates to false, the allow statement will become false. If OPA acting as PDP receives a request from a PEP, all allow statements are evaluated. If one allow statement (i.e., policy) is evaluated as true, the request is allowed. Therefore, a default Rego policy

is required denying every request by always evaluating as false. The rules presented in Listing 5.11 follow the structure used by the authorization policies created in the design phase. The order in which the rules occur follows the subject, action, object, and conditions pattern known from ABAC.

The left side of the boolean expressions presented in lines 3 to 6 contains the expected values. The right side contains the comparison values provided by the input variable. In OPA, the input variable provides the content of the HTTP request that is forwarded by the PDP. Depending on the PDP, the content (e.g., additional headers) and the structure of the request can vary. An example input produced by Envoy is documented in Appendix A.2. However, the content of the API request and the subject's access token must be present in the input. In this chapter, Envoy as an API proxy acting as a PDP is used as an example [EP-Doc]. In lines 4 and 5 of Listing 5.11, the path is retrieved from the request to evaluate the action and the object. This is due to a characteristic of gRPC, which structures gRPC services and RPCs in the HTTP path [Go-Pro]. Line 6 evaluates if the car is in the fleet manager's fleet by accessing the PIP using the values *vin* and *subject.sub* provided by the input.

### 5.3.1 Create Policy Implementation Structure

We propose an implementation structure presented in Figure 5.6 to further structure the implementation of the authorization policies and to reduce the duplication of code. The implementation structure consists of five main folders. The *policies* folder contains all policy implementations. For each authorization policy created in the design phase, a Rego file containing an allow statement is created. The folder *rules* contains all rules required by the authorization policies. The rules are further structured into the folders *subject*, *action*, *object*, and *condition*. The *data* folder contains the logic required to retrieve attributes from the PIP. Finally, the *utils* folder provides utility functionality, e.g., to access environment variables or to decode a JWT.

By utilizing the structure presented in Figure 5.6, the authorization policy presented in Listing 5.11 can be simplified. Each rule is implemented in a separate file. By using import statements, the authorization policy can include these rules. This result is a simplified allow statement inside the Rego file, as presented in Listing 5.12. The names of the files and the rules are written in the so-called snake_case, following the Rego style guide [Sty24]. Compared to the Rego policy presented in Listing 5.11, importing rules into the authorization policy provides several advantages: First, the rules are reusable. A rule can be used by multiple Rego policies, reducing the duplicated code. Second, each rule can be individually unit tested. An example is provided in Section 5.3.4. Finally, implementing the rules in separate files allows for the authorization policy implementation to become independent of the underlying technology. Two examples can be made here: First, as introduced before, if the PDP changes, the format of the input can change. In this case, the policy implementation will remain untouched while the rules are adapted to the new structure. Second, if the format of the access token changes, only the Rego rules for extracting and evaluating the access token must be changed.

```
authorization/rentalmanagement
├── policies
│   ├── default.rego
│   └── fleet_manager_list_car_rentals.rego
├── rules
│   ├── subject
│   │   └── subject_is_fleet_manager.rego
│   ├── action
│   │   ├── action_is_list_car_rentals.rego
│   │   └── action_is_rent_car.rego
│   ├── object
│   │   ├── object_is_rentals.rego
│   │   └── object_is_rentals_collection_service.rego
│   └── condition
│       └── car_in_fleet_of_fleet_manager.rego
├── data
│   └── fleet.rego
└── utils
    ├── jwk.rego
    └── jwt.rego
```

**Figure 5.6:** Policy Implementation Structure for Rego Policies

```
1 #---FleetManagerListCarRentals---
2 allow if {
3   subject_is_fleet_manager
4   action_is_list_car_rentals
5   object_is_rentals
6   car_in_fleet_of_fleet_manager
7 }
```

**Listing 5.12:** Example Authorization Policy Implementation

Therefore, for every new authorization policy to be implemented, the policy implementation structure is created. In the *policies* folder, a new Rego file including the allow statement is created. For every rule inside the authorization policy, a file with an empty rule is created in the respective folder. Each rule is subsequently referenced in the allow statement. If a rule already exists, the existing rule is referenced.

### 5.3.2 Implement Policy Rules

The next step is the implementation of the Rego rules. Listing 5.13 presents example implementations. As before, the name of the rules follows the snake_case, as proposed by the Rego style guide [Sty24]. In lines 1 to 4, the presence of the role *fleetmanager* is evaluated. To retrieve the roles from the subject, the access token must be retrieved from the incoming request. This is described by the example of a JWT in lines 20 to 28. JWTs consist of three parts that can be decoded using built-in Rego functionality (line 22). The decoded payload contains the required OIDC claims, such as a role or sub. Before the JWT can be decoded, the token must be extracted from the request (lines 24 to 28).

```
1  # subject_is_fleet_manager.rego
2  subject_is_fleet_manager {
3      "fleetmanager" in subject.roles
4  }
5  # action_is_list_car_rentals.rego
6  action_is_list_car_rentals {
7      "ListCarRentals" == input.parsed_path[1]
8  }
9
10 # object_is_rentals.rego
11 object_is_rentals if {
12   "rentalmanagement.RentalsCollectionService" == input.parsed_path[0]
13 }
14
15 # car_in_fleet_of_fleet_manager.rego
16 car_in_fleet_of_fleet_manager if {
17   data.car_in_fleet_of_fleetmanager(input.parsed_body.vin, subject.sub)
18 }
19
20 #utils jwt.rego
21 subject := payload {
22     [_, payload, _] := io.jwt.decode(bearer_token)
23 }
24 bearer_token := t {
25     v := input.http_request.headers.authorization
26     startswith(v, "Bearer ")
27     t := substring(v, count("Bearer "), -1)
28 }
```

**Listing 5.13:** Implementation of Rego Rules

Lines 6 to 8 present the evaluation of the action. Analogous, lines 10 to 13 evaluate the object. Lines 15 to 18 show the implementation of the condition which evaluates if a car is in the fleet manager's fleet. To evaluate this condition, the PIP of the microservice FleetManagement must be accessed. The

connection to the PIP is documented in the next section. Inside the data folder, for each object that requires access to a PIP, a file is created. Using functions written in Rego, the required values are returned. For instance, the function *data.car_in_fleet_of_fleetmanager(vin, subjectID)* takes a *vin* and a *subjectID* as input parameters and returns a true/false result.

### 5.3.3 Retrieve Attributes from PIP

An essential aspect required for the evaluation of authorization policies is the retrieval of attributes. An important aspect here is the topicality of the attributes. Otherwise, authorization decisions are based on old and possibly invalid attributes. Therefore, the attributes must be retrieved from a PIP. By default, OPA provides built-in functionality, that allows Rego to perform HTTP requests. If the PIP provides a RESTful interface, the attributes can be retrieved through an HTTP request. An example is presented in lines 2 to 5 in Listing 5.14. If the request is successful (line 6), the car is in the fleet manager's fleet. This functionality can be exposed through a function such as *data.car_in_fleet_of_fleetmanager(vin, subjectID)* which returns true or false (lines 1 and 7).

```
1 car_in_fleet_of_fleetmanager(vin, subjectID) :=  if {
2   response := http.send({
3         "method": "GET",
4         "url": sprintf("https://PIP-FleetManagement/fleetmanager/%s/car/%s
              ", subjectID, vin)
5     })
6     response.status_code == 200
7 }
```

**Listing 5.14:** Rego Rule to Retrieve Attributes Via HTTP Request

The retrieval of attributes depends on the selected technology providing the PIP. In Section 7.3.1, an extension to OPA is presented, which can directly access a relational database such as PostgreSQL.

### 5.3.4 Testing Authorization Policies

Similar to the business logic of the microservice, the authorization logic should be thoroughly and systematically tested. While this is considered future work, this section provides a brief introduction to policy testing. The procedure depends on the used policy language. OPA provides a utility to develop and execute Rego tests [OP-Do]. Listing 5.15 presents an example test for the Rego rule *fleet_manager_assigned_to_fleet*. Tests are also structured as a Rego rule, that must evaluate to true or false depending on the expected outcome for the test to be considered successful. For OPA to execute the tests, the test name must begin with *test_*. First, the test is set up by initializing test data in lines 2

```
1 test_fleet_manager_assigned_to_fleet if {
2   mock_fleetID := "1"
3   mock_token := {"sub": "9a6e63f9-be1b-453c-b67e-9b4297580236"}
4   mock_response := {"status_code": 200, "body": "9a6e63f9-be1b-453c-b67e-9
        b4297580236"}
5
6   fleet_manager_assigned_to_fleet with input.parsed_body.fleetId as
        mock_fleetID
7     with data.authn.token as mock_token
8     with http.send as mock_response
9 }
```

**Listing 5.15:** Unit Test for Rego Rule "fleet_manager_assigned_to_fleet"

to 4. The *fleetID* is initialized to 1 (line 2). In line 3, the content of a JWT access token is created with a mocked identifier of a subject. The same subject identifier is used in the mocked response of the PIP depicted in line 4. With the initialized test data, the tested Rego rule or policy is referenced. The existing data is overwritten using the statement *with*. Since the subject identifier in the access token and the PIP response are equal, the test succeeds.

Unit tests for authorization policies and rules as presented in Listing 5.15 are only one aspect of policy testing [HK+17]. While the testing of authorization is considered future work (see Section 9.2), the authorization policies should be thoroughly and systematically tested to assure a high quality of the policy implementations. This includes dedicated steps in the development process, such as the creation of test data or a test-driven implementation of tests. Furthermore, additional test types such as integration tests or end-to-end tests must be considered in the context of the overall integration into a microservice-based application. In addition, the application should not be deployed if the authorization policy tests fail (see Section 7.4.1). Otherwise, a subject might gain access to a resource they are not authorized to.

## 5.4 Summary

Figure 5.7 presents a detailed overview of the process to develop authorization policies introduced in this chapter. The process spans through the phases of analysis, design, and implementation and test. In total, three authorization artifacts, one in each development phase, are introduced. For each development phase, a sub-process is introduced to derive the respective authorization artifact.

The goal of the analysis phase is the derivation of authorization requirements. Authorization requirements specify what must be authorized. The foundation for the authorization requirements are the functional requirements. The subject, object, action, and conditions are identified from the functional

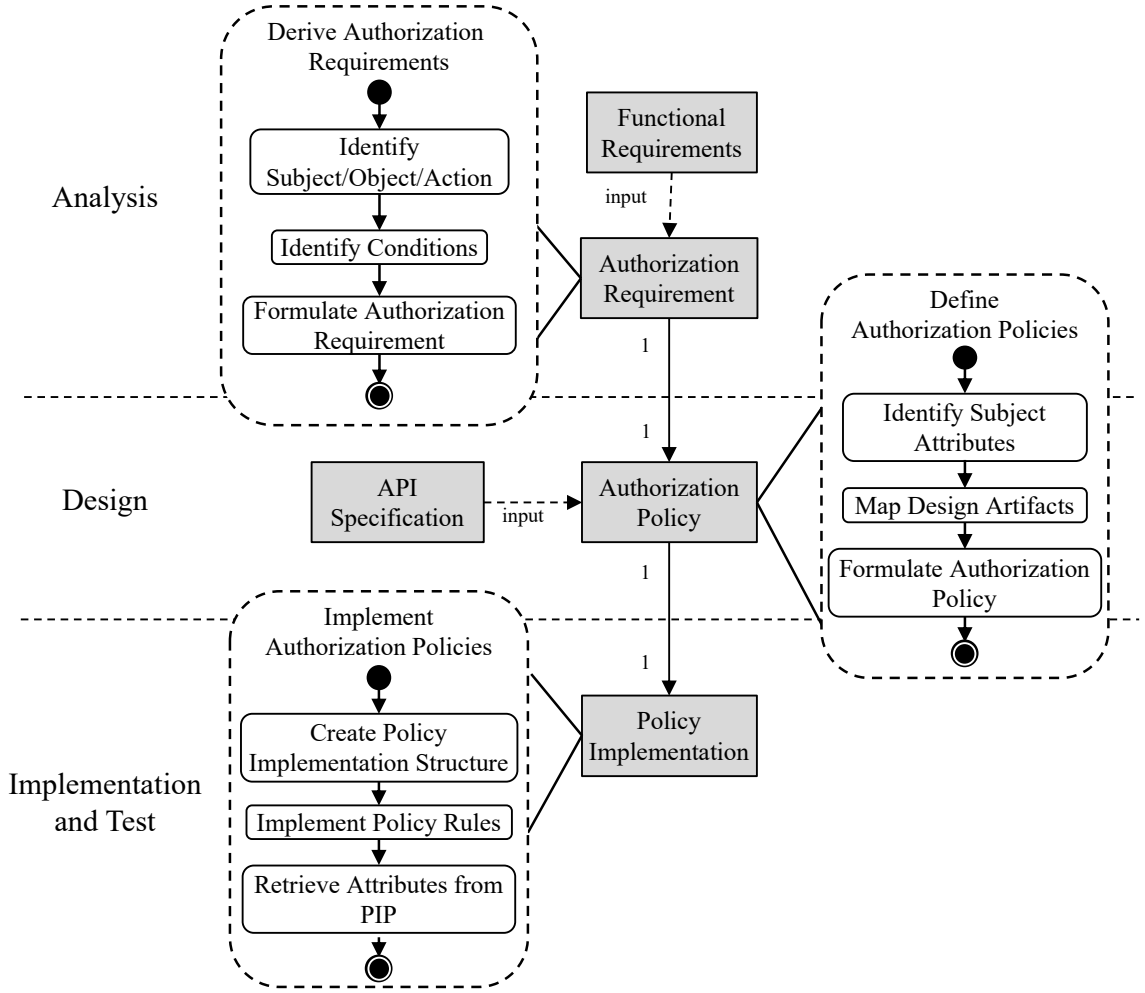


**Figure 5.7:** Detailed Overview of the Authorization Policy Development

requirements are subsequently formulated into an authorization requirement. A template is used to structure the authorization requirements. The derivation of authorization requirements is limited by the employed ABAC model (see Section 2.5), which only considers one subject performing an action on an object under a given set of conditions at a given time. The authorization requirements are transformed into authorization policies in the design phase. With the help of the API specification of the microservices, the attributes are identified, and the authorization policy is formulated. The authorization policy is documented using a template and is independent of a specific policy language. The conditions are specified using boolean statements. In the implementation and test phase, the authorization policies are implemented in a policy language. By using Rego, this thesis provides a systematic and structured implementation.

The authorization artifacts in this chapter concentrated on the authorization of a user, i.e., subject, interacting with the microservice-based application. This is due to the use of functional requirements

as the foundation for the authorization policies. The next chapter investigates the authorization between microservices that occurs as a result of the user interacting with the microservice-based application. Therefore, artifacts from this chapter (i.e., authorization requirements) are considered to make the Service-to-Service (S2S) authorization fine-grained.

# 6 Service-to-Service Authorization

The previous chapter introduced the development of authorization policies from the perspective of a user interacting with a microservice-based application. Since a microservice-based application typically consists of a set of multiple distributed microservices, there are also requests between microservices. These Service-to-Service (S2S) requests can be necessary, e.g., to retrieve further data to provide the business logic of a microservice. It should be noted that the interaction between microservices leads to the coupling of microservices, which should be carefully considered when designing a microservice-based application [Ne15].

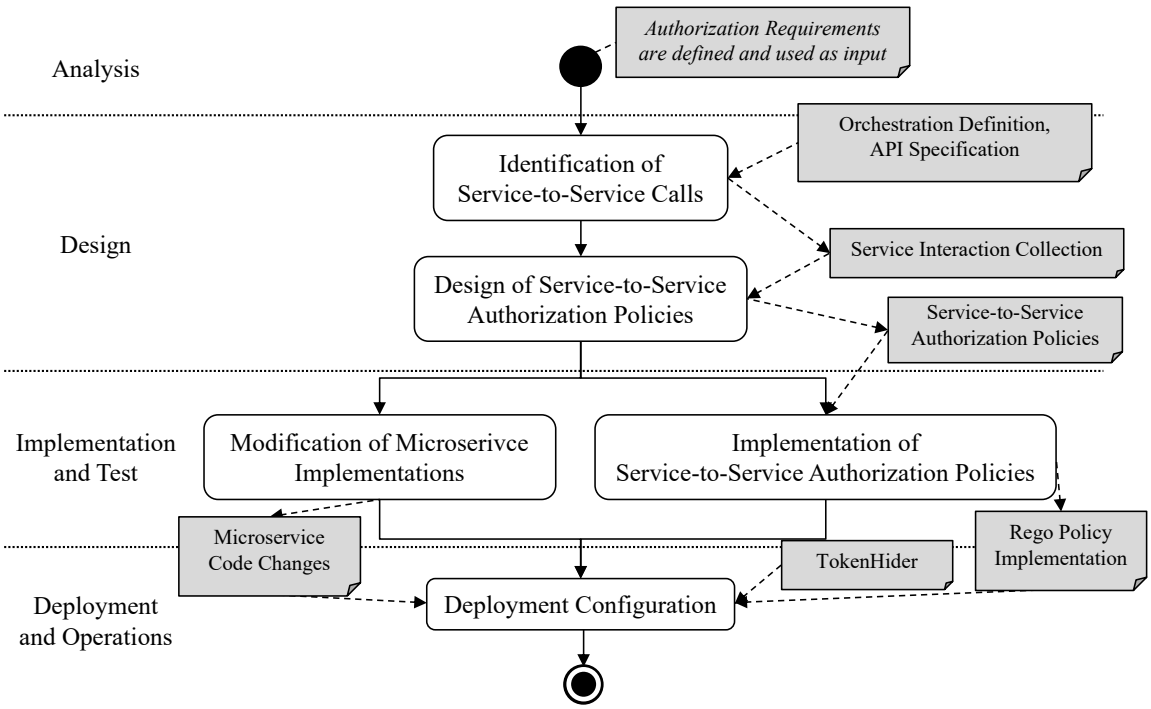


**Figure 6.1:** Overview of Service-to-Service Development Process

The consideration of authorization for S2S interactions is the focus of this chapter. With the migration to zero trust, the physical distance between microservices does not provide a sufficient security mechanism [TU+21; RB+20]. For instance, if a microservice in a Kubernetes namespace is compromised, the compromised microservice might be able to retrieve data from other microservices in the same namespace. Therefore, additional security measures (e.g., identity propagation) are required to authorize every request. Existing approaches towards S2S authorization such as Log2Policy [XZ+23]

or AutoAmor [LC+21] use logs files or source code analysis respectively to create S2S authorization policies. However, these authorization policies can be considered coarse-grained as they limit access to resources to a set of microservices but do not incorporate additional attributes (e.g., from the user). In addition, these approaches take place exclusively at the implementation and deployment level and do not include any analysis or design artifacts. In this chapter, we elaborate the systematic implementation of S2S authorization policies, which is contribution C2 of this thesis. The goal is to create S2S authorization policies that are fine-grained. An overview of the proposed process is presented in Figure 6.1. The process is structured into the phases of analysis, design, implementation and test, and deployment and operations.

In the analysis phase, the authorization requirements established in Section 5.1 define what the subject is allowed to perform. These authorization requirements should hold for the application as a whole. Since the interaction between microservices is a result of the design of the microservice-based application, these authorization requirements should also apply to S2S requests. A brief overview of the analysis phase using the authorization requirements as input for S2S authorization is introduced in Section 6.1.

The predominant aspects of S2S authorization take place in the design phase, which is described in Section 6.2. In the design phase, the decision to create a microservice-based application has been made. This includes the microservice cut, i.e., the business logic that is provided by each microservice. The first step is the identification of the S2S interaction in the microservice-based application. The basis for this is the software architecture (e.g., UML component diagram), which depicts the interaction on a high level. Thus, further orchestration definitions are required to understand how the microservices interact with each other. In addition, the Application Programming Interface (API) specifications are required to recognize the API requests performed in the S2S interaction. The result of the initial step is a service interaction collection, which provides an overview of all S2S calls. Similar to the authorization policies created in Chapter 5, authorization policies for the S2S interaction are created. Again, these S2S authorization policies are independent of a policy language.

Section 6.3 presents the implementation aspects of S2S authorization. The implementation consists of two aspects: First, the implementation of the previously created S2S authorization policies in the Rego policy language. This is similar to the implementation of the authorization policies and allows reusing Rego rules created previously. A special characteristic of the S2S authorization Rego policies is the evaluation of the source microservice initiating the S2S request. Second, while requirement *R5 - Externalized Authorization* requires externalized authorization which implies no integration of authorization logic inside the microservice, S2S authorization requires the microservice to support identity propagation [YB18] (see Section 2.6.1). Identity propagation allows identifying the subject initiating a S2S request chain by propagating the subject's credentials through the S2S requests. To perform fine-grained S2S authorization, understanding who, i.e., which human user, initiated the S2S request is essential. This requires modifications to the code base of the microservice.

Finally, Section 6.4 discusses the deployment of S2S authorization in a microservice-based application. We introduce the TokenHider, which allows an access token to be withheld from a microservice. This prevents a compromised microservice from misusing the access token for unauthorized access to external services.

## 6.1 Analysis

As presented in Figure 6.1, the engineering of S2S authorization policies starts in the design phase. This is because the software architecture does not exist in the analysis phase. Instead, the software architecture is created in the design phase [Wa24]. This includes design decisions that introduce communication between different microservices. For the S2S interaction that is the result of user interaction with the application, the authorization requirements created in the previous chapter apply, since the authorization requirements are created independent of the employed software architecture.

Therefore, the previously created authorization requirements should also hold for the S2S requests and are thus used as input for the creation of S2S authorization policies. Listing 6.1 presents an authorization requirement for the use case *Remove Car from Fleet*, which is used as an example in this chapter. In this use case, the fleet manager is allowed to remove a car from an object fleet if two conditions are fulfilled. First, the fleet manager must be assigned to the targeted *Fleet*. Second, the car which should be removed must also be in the fleet of the fleet manager. Further non-functional requirements towards the technical realization of S2S authorization are described in Section 7.1.

```
1 ---FleetManagerRemoveCarFromFleet---
2 subject FleetManager is allowed to perform
3 action RemoveCar on
4 object Fleet if
5 condition FleetManagerAssignedToFleet and
6 condition CarInFleet
```

**Listing 6.1:** Authorization Requirements "Remove Car from Fleet"

## 6.2 Design

In the design phase, the design decision to use the microservice architecture is made. This leads to the business logic to be realized by multiple microservices. Each microservice is self-contained and provides an excerpt of the business logic (e.g., a set of use cases) through an API. If a microservice requires additional data that is handled by another microservice, the APIs of the respective microservices must provide the functionality. This requires an orchestration between the microservices [DG+17;

Ne15]. In the CarRentalApp example, to remove a car from a fleet, the microservice FleetManagement must first verify if the microservice RentalManagement has no more rentals for the car. Hence, an orchestration definition is required. This allows to identify the order in which requests occur between microservices.

In the authorization context, the order in which the S2S requests occur is relevant. Depending on the request order, information that could be used for authorization is available or has already been updated. For example, when deleting a car from FleetManagement before it has been deleted from RentalManagement, the information of the related fleet is no longer available for authorization of a delete request at RentalManagement. However, this behavior can be intentional, depending on the design of the microservice-based application. Different transaction patterns can be considered when designing an application. To realize S2S authorization, the transaction patterns and thus the order in which requests occur must be analyzed.

This chapter focuses on the authorization of S2S requests that are the result of the interaction of a user with the microservice-based application. However, the authorization of other S2S requests that are the result of design decisions are briefly introduced in Section 6.5. Because these S2S requests do not have a respective authorization requirement, the granularity will be limited (e.g., there is no involved subject). Nonetheless, the approach presented can be employed with minor restrictions.

### 6.2.1 Identification of Service-to-Service Calls

To create fine-grained authorization policies (R3), the occurring S2S calls must be identified first. Therefore, the software architecture can be examined. For instance, the software architecture of CarRentalApp depicted in Figure 1.3 implies a connection between the microservice FleetManagement and RentalManagement. This leads to the questions of which requests occur and in which order do these requests occur. We propose that all identified S2S requests occurring in a microservice-based application are collected in a table. Figure 6.2 presents an overview of the targeted table using use cases as the underlying functional requirement. The subject is the initial actor that executes a specific use case. As presented in Chapter 5, for each functional requirement, an authorization requirement is created. These authorization requirements define what a user is allowed to do under a set of conditions. This authorization requirement is independent of the microservices. Subsequently, the authorization requirement should also be valid for S2S requests and is therefore included in the table. Next, the involved microservices are recorded. As proposed by Li et al. [LC+21], the microservice starting the S2S request on behalf of the subject is defined as the source microservice, and the microservice receiving the S2S request is considered as the target microservice. Finally, the request itself is analyzed by identifying the action (e.g., REpresentational State Transfer (REST) or Remote Procedure Call (RPC) operation) and the object. Analogous to the authorization introduced in Chapter 5, the object is considered to be the API endpoint.

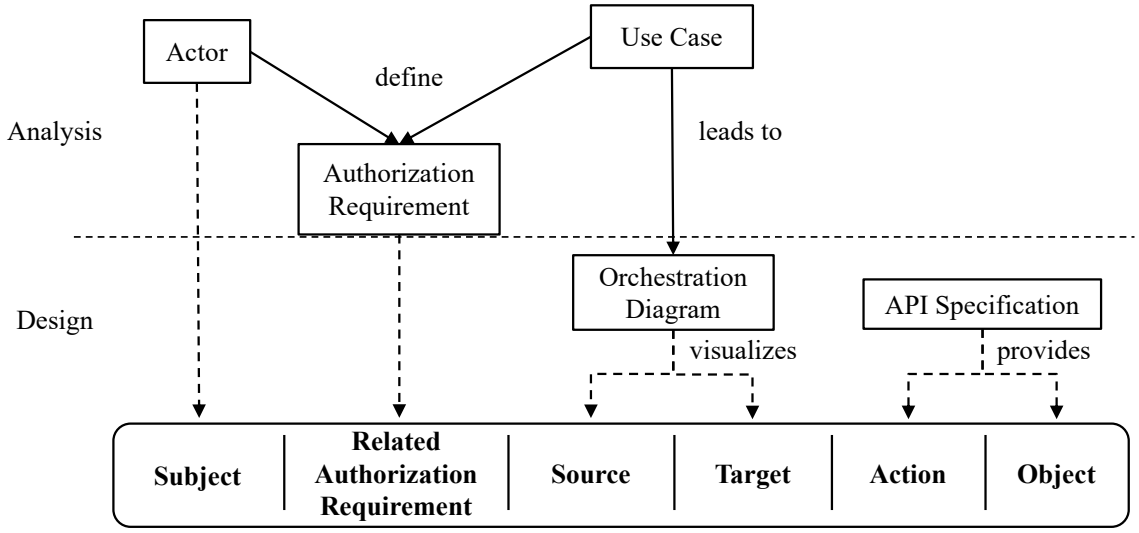


**Figure 6.2:** Identification of Service-to-Service Requests

An example of the resulting S2S interaction table is presented in Table 6.1. For the use case *Remove Car from Fleet*, the related authorization requirement is depicted in Listing 6.1. The subject *FleetManager* creates a request from the microservice FleetManagement (source) to the microservice RentalManagement (target). The request is a gRPC request including the *RPC RemoveRentableCar* on the *RentableCarsCollectionService*.

| **Subject** | **Related Authorization Requirement** | **Source** | **Target** | **Action** | **Object** |
|---|---|---|---|---|---|
| Fleet-Manager | FleetManager-RemoveCar-FromFleet | Fleet-Management | Rental-Management | Remove-RentableCar | RentableCars-Collection-Service |

**Table 6.1:** Exemplary Service Interaction Collection

To identify the orchestration in which the S2S requests occur, an orchestration diagram is used. This has also been proposed by Schneider [Sc24]. Orchestration diagrams are inspired by the Business Process Execution Language (BPEL) [Ju06] and Service-oriented Architecture Modeling Language (SoaML) [EB+11]. Schneider uses an Unified Modeling Language (UML) activity diagram to model the orchestration between microservices. However, a simple UML sequence diagram will also display the interactions between the subject and the respective microservices.

Figure 6.3 presents the orchestration diagram for the *RPC removeCarFromFleet*. The start is the execution of the *RPC removeCarFromFleet* with the given Vehicle Identification Number (VIN) as a parameter by the microservice FleetManagement. The microservice performs initial checks, e.g., to validate if the car with the VIN exists. Subsequently, the microservice FleetManagement performs the S2S request *RPC removeRentableCar* with the given VIN as a parameter to the microservice

RentalManagement. If there are still rentals for the given VIN, the microservice returns an error. Otherwise, the car is removed from RentalManagement and a success message (e.g., HTTP code 200) is returned to FleetManagement. Depending on the response, FleetManagement either returns an error or removes the car from its database and returns a success message.

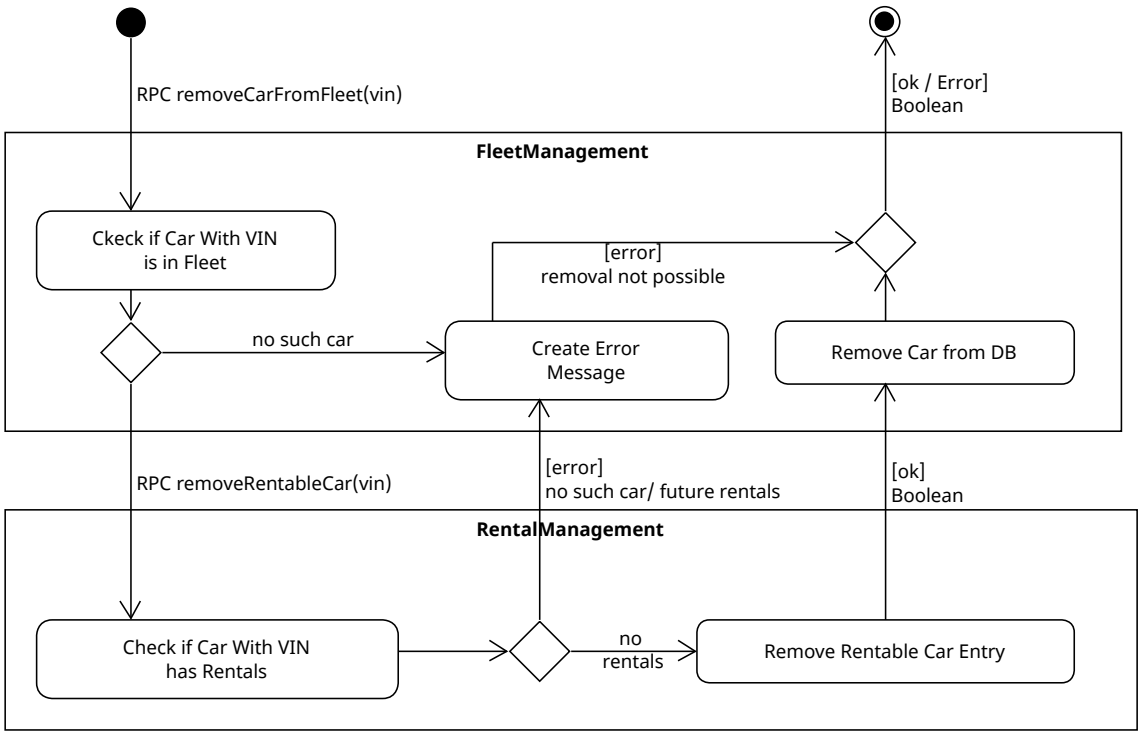


**Figure 6.3:** Orchestration Diagram for the Use Case "Remove Car From Fleet"

From the orchestration diagram, the API endpoints can be extracted. The API specification provides the attributes that can be used for authorization. For instance, the *RPC removeRentableCar* uses the VIN to identify the car that should be removed. This VIN is also used by the microservice FleetManagement to identify the car.

### 6.2.2 Design of Service-to-Service Authorization Policies

After identifying the S2S requests, the S2S authorization policies are designed. Similar to the authorization policies presented in Chapter 5, the S2S authorization policies are independent of the policy language. The structure of the S2S authorization policies is similar to the authorization policies. The template for S2S authorization policies is presented in Listing 6.2. The S2S authorization policies are grouped by the target microservice. As an identifier for the authorization policy, the name of the target microservice and the authorization requirement are used (line 1).

The object is the API endpoint of the target microservice (line 4). The action depends on the employed API paradigm (line 3). For a RESTful API, the action represents the HTTP operation. In comparison, if a gRPC API is used, the action represents the respective RPC. To identify the source of the S2S request, an additional condition is added (line 5). This is necessary to identify the source microservice and to prevent that any compromised microservice can perform the S2S request. There are several options to identify the source microservice, which are further elaborated in the next section. Besides the source microservice, the subject responsible for the initiation of the S2S request is required (line 6). Finally, additional conditions from the authorization requirement are included (line 7). These conditions are similar to the authorization policy introduced in Chapter 5.

```
1 ---<Target>-<RelatedAuthorizationRequirement>---
2 service can perform
3 action <Action> on
4 object <Object> if
5 source is <Source> and
6 <Subject> in subject.roles and
7 <Conditions>
```

**Listing 6.2:** S2S Authorization Policy Template

An example of the S2S authorization policy for the use case *Remove Car From Fleet* is presented in Listing 6.3. The subject, action, object, and source can be adopted (lines 2 to 6) from the service interaction collection. The conditions must be adopted from the authorization requirement. The authorization requirement presented in Listing 6.1 requires that the fleet manager is assigned to the fleet and that the car to be removed is in the fleet manager's fleet. However, since this request is a S2S request, the RentalManagement microservice does not know anything about Fleets. The microservice FleetManager is responsible for the fleet entity. Since the API request contains the VIN of the car, the Policy Information Point (PIP) of the microservice FleetManagement can be accessed to retrieve the necessary information. Hence, the conditions verifying that the car is in the fleet of the fleetmanager are added to the authorization policies (lines 7 and 8). Accessing the attributes through the PIP from other microservices allows creating fine-grained S2S authorization decisions. This is possible for longer S2S request chains as long as there are identifiers such as the VIN to retrieve the respective attributes. However, we identify two cases in which the granularity of the S2S authorization is limited.

First, there is a lack of attributes that can be used for authorization. This is the case if the API of the targeted microservice does not contain an identifier that is related to the initial request. For example, if the microservice RentalManagement creates an additional S2S request to a notification microservice which informs about a changed collection of rentable cars, the S2S request does not contain a VIN or

```
1 ---RentalManagement-FleetManagerRemoveCarFromFleet---
2 service can perform
3 action RemoveRentableCar on
4 object RentableCarsCollectionService if
5 source is FleetManagement and
6 fleetmanager in subject.roles and
7 Fleet.fleetManager == subject.sub and
8 Fleet.cars contains input.vin
```

**Listing 6.3:** Example S2S Authorization Policy for "Remove Car From Fleet"

a fleetID. Hence, besides the evaluation of the initiating subject, the action, the object, and the source microservice, there are no additional conditions.

Second, the transaction pattern prevents the retrieval of attributes. Several transaction patterns can be applied to distributed systems, such as microservice-based applications [Ru18]. For example, the Saga pattern [GS87] specifies that attributes are modified first, before the next service is called. If the following service call throws an error, the operations are rolled-back to the last consistent state. Another pattern is the Two-Phase Commit (2PC) protocol [Kl17]. If the 2PC protocol is applied, the microservices perform the S2S requests in the first phase. Instead of committing the transaction to, e.g., the database, the microservice prepares the commit. Only if all S2S requests are performed successfully, the transaction is committed. Depending on the used transaction protocol, the attributes relevant to authorization have already been updated or are outdated. For example, if the Saga pattern is used, FleetManagement deletes the car from the fleet before performing the S2S request to RentalManagement. In this case, RentalManagement cannot evaluate if the car is in the fleet of the fleet manager. Subsequently, the S2S authorization policy would be less fine-granular. However, a similar problem can occur with the 2PC protocol. If the fleet manager adds a new car to its fleet, the FleetManagement microservice would first perform a S2S request to add the car to the rentable cars of RentalManagement. Since the car has not yet been committed to the fleet manager's fleet, the condition to check if the car is in the fleet of the fleet manager cannot be evaluated in the S2S request. Because this depends on design decisions, we cannot provide a general solution to this problem. If fine-granular S2S authorization is desired, the sequence of transactions must be considered, when designing the orchestration of a microservice-based application.

## 6.3 Implementation and Test

Moving from the design phase to the implementation and test phase, the S2S authorization policies designed in the previous section must be implemented. Generally, the implementation of these S2S authorization policies in a policy language such as Rego is similar to the authorization policies

presented in Chapter 5. However, for the implementation of S2S authorization policies, there are key differences that must be considered. First, the storage of the S2S authorization policies. Second, the evaluation of the additional conditions that determine the source microservices starting the S2S request. Third, performing requests to PIPs to retrieve attribute data. Furthermore, to realize S2S authorization, the Policy Decision Point (PDP) of the target microservice must know the subject that started the initial request. This requires modifications of the microservice which are introduced in Section 6.3.2.

### 6.3.1 Implementation of Service-to-Service Authorization Policies

The implementation of the S2S authorization policies in Rego is similar to the concepts in Section 5.3. Listing 6.4 presents the Rego policy for the use case *Remove Car from Fleet*. The S2S authorization policy is structured inside an allow statement (lines 2 to 8) in a dedicated Rego file. In the S2S authorization policy implementation, each rule is located in a separate Rego file, which can be imported.

```
1 #fleetmanagement_remove_rentable_car.rego
2 allow if {
3     action_is_remove_rentable_car
4     object_is_rentable_cars
5     source_is_am_fleetmanagement
6     subject_is_fleet_manager
7     car_in_fleet_of_fleet_manager
8 }
```

**Listing 6.4:** S2S Authorization Policy Implementation for "Remove Car From Fleet"

Figure 6.4 presents the structure of the Rego authorization policies for the microservice Rental-Management. Since the S2S authorization policies must be enforced at the target microservice, the implemented Rego policies must be stored at the target microservice. For the S2S request *RPC removeRentableCar*, the policy is stored in the policies folder next to the other authorization policies in the the policy storage of the microservice RentalManagement. In addition, a folder called *source* is added to the folder structure. This folder contains the rules that identify the source microservice of a S2S request.

The rules inside an allow statement can be reused from other (user) authorization policies. This allows to perform unit tests on a rule level (see Section 5.3.4). For example, the rule *object_is_rentable_cars* is also required by the use case *List Rentable Cars* which can be performed by a fleet manager. The evaluation of the subject initiating the request (i.e., fleet manager) is similar to the authorization policies. An access token (e.g., JSON Web Token (JWT) format) identifying the subject must be

```
/policies/rentalmanagement/
├── utils
├── policies
│   ├── default.rego
│   └── fleetmanagement_remove_rentable_car.rego
├── rules
│   ├── subject
│   │   └── subject_is_fleet_manager.rego
│   ├── action
│   ├── object
│   ├── condition
│   └── source
│       └── source_is_fleetmanagement.rego
└── data
    └── fleet.rego
```

**Figure 6.4:** Structure for Authorization Policies

present. This is evaluated in the rule *subject_is_fleet_manager* (line 6 of Listing 6.4). However, since the S2S request is performed by a microservice, the respective subject access token is not necessarily available in the S2S request. Therefore, the microservice must support the propagation of the subject's identity [YB18]. The necessary changes are introduced in Section 6.3.2.

Compared to the implementation of authorization policies, the rule regarding identification and evaluation of the source microservice performing the S2S request is added. In the example presented in Listing 6.4, the rule *source_is_fleetmanagement* (line 5) is responsible for the evaluation of the source microservice. The trust between microservices is a challenge when it comes to authentication and authorization in the microservice architecture [AC22]. There are several options to establish the trust, i.e., identifying that a microservice is who it claims to be. In this section, API keys and mutual Transport Layer Security (mTLS) are presented as possible options solutions.

**Use of API Keys** API keys are a common mechanism to allow access to an API [De17]. The API key is typically issued by a service provider. To access the API, the API key is sent alongside other data, e.g., as a header, in an (encrypted) HTTP(S) request. The service provider can subsequently evaluate if the API key is valid and allow access to the API.

The concept of API keys can also be utilized to authorize S2S requests [De17]. In the example of the microservices FleetManagement and RentalManagement, RentalManagement can create an API key for FleetManagement. The microservice FleetManagement has to include the API key in the S2S

```
#source_is_am_fleetmanagement.rego
source_is_am_fleetmanagement if {
    input.http.headers["api-key"] == env("API_KEY_ENV")
}
```

**Listing 6.5:** Rego Rule Evaluating API Key

requests to RentalManagement. The Rego authorization policy can then evaluate the API key. An example is presented in Listing 6.5. The rule *source_is_fleetmanagement* compares the content of the HTTP header *api-key* from the S2S request with the environment variable *API_KEY_ENV* that is available to the PDP Open Policy Agent (OPA). This requires that the API key must be known by the microservice FleetManagement and the PDP.

While the use of API keys has a low complexity, there are security risks involved in using API keys that should be addressed [Lu14]. First, API keys should never be stored in the code. Instead, API keys should always be retrieved from environment variables. This prevents the leakage of API keys that could compromise the S2S authorization. Second, it is a common practice that API keys have a limited lifetime and are frequently rotated. This means that API keys are replaced at regular time intervals (e.g., daily). If an API key has been leaked, the key rotation will limit the possible access to the remaining time interval the API key is valid.

**Use of Mutual TLS** Another option to ensure that the traffic between microservices (i.e., a client and a server) is authenticated is the use of mTLS. The National Institute of Standards and Technology (NIST) proposes the use of mTLS to provide trust in a microservice network [Ch19]. mTLS is part of the Transport Layer Security (TLS) protocol, which is the most commonly used cryptographic protocol [KP+13]. Currently in web services, version 1.3 [Er18] is used to provide a secure connection of HTTPS over TLS. The web server has a certificate issued by a trusted Certificate Authority (CA). The client trusts this certificate, and TLS allows a secure connection to be established (through different handshakes). If mTLS is used, the client also requires a certificate from a CA which is trusted by the server. The client has to provide the certificate in the HTTPS request. The server subsequently verifies the certificate and a secure end-to-end encrypted channel has been established.

If the architecture proposed in Chapter 7 is used, the Policy Enforcement Point (PEP) will intercept the request from the source microservice and forward the request to the PDP. For example, if Envoy is used as a PEP, the client certificate is forwarded to the PDP (i.e., OPA). However, Envoy can already verify, that the request comes from a client that has a certificate from a trusted CA. Assuming that every microservice of a microservice-based application has a respective certificate, Envoy could not differentiate if a request stems from a microservice that is allowed to perform a request or not. Thus, if

a microservice is compromised, it could perform requests to other microservices. However, since the PEP forwards the request to the PDP which contains the client certificate, the content of the certificate can be verified by the PDP to allow a fine-grained authorization.

```
1 #source_is_am_fleetmanagement.rego
2 source_is_am_fleetmanagement if {
3     clientCert = input.http.headers["X-Forwarded-Client-Cert"]
4     #"By=...;Hash=...;Subject=../CN=FleetManagement;URI=..."
5     source = split(split(clientCert, ";")[0], "=")[2]
6     source == "../CN=FleetManagement"
7 }
```

**Listing 6.6:** Rego Rule for Evaluation of mTLS Certificate

In Listing 6.6, the Rego rule *source_is_am_fleetmanagement* is presented ins lines 2 to 7. First, the client certificates are retrieved from the HTTP headers. The commonly used certificates follow the X.509 format [HP+99]. For instance, Envoy forwards the subject field from the certificate to OPA. An example of the forwarded client certificate is presented in line 4. The Rego rule subsequently extracts the subject field (line 5) and compares the result with the expected subject (line 6). If the comparison is successful, the source microservice has correctly been identified. This allows the S2S authorization policy to ensure that the request stems from the correct source microservice, assuming that the private and public keys of the client certificates have not been leaked, and the trusted CA has not been compromised.

### 6.3.2 Modification of Microservice Implementations

To realize the S2S authorization policy implementation presented in Listing 6.4, the PDP of the target microservice must be able to identify the subject that is responsible for initiating the S2S request chain [CB+21]. This allows to access the subject attributes in the S2S authorization policies. For instance, the car can only be deleted from the microservice RentalManagement, if the subject owns the fleet that the car is part of. This requires that the microservice includes the information of the subject in the S2S requests. This concept is also known as identity propagation (see Section 2.6.1) [YB18; CB+21]. While Microservice Authorization Framework (MAF) targets externalized authorization, modifications to the code base of a microservice are necessary to support identity propagation for fine-grained S2S authorization.

Identity propagation can be understood as part of the more general concept of context propagation [MF18]. In distributed systems, such as microservices, context propagation allows defining a context that can be traced through the chain of requests inside the distributed system. A popular example is

```
1  // Receives RPC from fleet manager
2  func (controller FleetController) RemoveCarFromFleet(ctx context.Context,
       req *pb.RemoveCarFromFleetRequest) (*pb.RemoveCarFromFleetResponse,
       error) {
3    ...
4    md, _ := metadata.FromIncomingContext(ctx)
5    header := md.Get("authorization")[0]
6
7    ctx = context.WithValue(ctx, "authorization", header)
8    ...
9  }
10
11 // Sends an RPC request to remove a rentable car
12 func (c *RentableCarsCollectionClient) RemoveRentableCar(ctx context.
       Context, vin model.Vin) (bool, error) {
13   ...
14     // Propagate context
15   md, _ := metadata.FromIncomingContext(ctx)
16   newContext := metadata.NewOutgoingContext(context.Background(), md)
17
18   resp, err := c.client.RemoveRentableCar(newContext, req)
19   ...
20 }
```

**Listing 6.7:** API Controller Providing Identity Propagation

OpenTemetry [Ope-Doc], a tool that allows to trace the S2S requests. This is achieved through the inclusion of an identifier. Tracing requests in a microservice-based application allows to, e.g., monitor traffic on a fine-granular level, identify bottlenecks, or security issues.

Listing 6.7 presents an example implementation of identity propagation in the Go programming language. Lines 1 to 9 present the function that is provided by the API controller. If the *RPC removeCarFromFleet* is performed to the microservice FleetManagement, the function in line 2 is invoked. Inside the microservice, the business logic is executed which finally leads to the microservice FleetManagement performing the gRPC request to the microservice RentalManagement in the function *RemoveRentableCar* depicted in lines 12 to 20. To support identity propagation, the context of the gRPC request must be forwarded through the microservice's business logic. The Go programming language provides the context package [Go-Con] which is a key-value store that allows to transfer data between different boundaries of the microservice's code.

In line 4, the metadata from the gRPC request is extracted. This includes the HTTP headers of the request. From the metadata, the authorization header is retrieved in line 5. This results in the access token from the fleet manager, e.g., a JWT. If another header is used to provide the subject identity, the name of the header must be chosen respectively. Finally, the access token is added to the key-value store of the context object (line 7). The context object *ctx* is subsequently passed through the business

logic. If the function *RemoveRentableCar* is reached (line 12), a new metadata object is created based on the context (lines 15 and 16). The metadata object is then included in the S2S request to the microservice RentalManagement (line 18). A similar approach is also possible in other programming languages, such as Java and the Spring Framework.

## 6.4 Deployment and Operations

To enforce the S2S authorization policies in a microservice-based application, the architecture presented in Chapter 7 can be used. Using the identity propagation, the identity of the subject is propagated through the different microservices. If a single microservice of a microservice-based application is compromised, the microservice has access to the access token of the subject. This allows the microservice to misuse the access token by accessing other services in the name of the subject. This problem is also called powerful-token problem [NJ+18] or the confused deputy problem [Ha88]. Hence, identity propagation also introduces a security risk into the microservice architecture [VK+23]. To counteract this problem, companies such as Netflix exchange the access token of a subject to an internal access token [Net-Bui]. The internal access token limits the potential damage resulting from a leaked access token. Exchanging the access token to an internal access token is also proposed by the OWASP Application Security Verification Standard by the Open Web Application Security Project (OWASP) foundation [OW21]. Unfortunately, the component responsible for exchanging the access token to an internal token must be trusted to not misuse the access tokens. In addition, developers must be trusted with the secure handling of access tokens and the correct implementation of identity propagation.

To overcome the potential misuse of an access token, we propose the TokenHider component. The TokenHider allows the microservice to provide business functionality in the absence of authorization mechanisms or access tokens. For instance, the gRPC request *removeCarFromFleet* for the microservice FleetManagement should also work without any authentication or authorization mechanisms. Therefore, besides the identity propagation required for S2S authorization, the microservice does not need to have access to the access token. The goal of the TokenHider is to remove the access token from the microservice. If the architecture presented in Section 7.2.1 is used, the access token passes the PEP, the PDP, and the microservice. Following the principle of separation of concerns, the TokenHider removes the access token from the request after the PEP has enforced the authorization decision. This ensures that the token is only available where it is needed for the execution of authorization logic (i.e., PEP, and PDP).

An overview of the TokenHider is presented in Figure 6.5. If the TokenHider is used, the (authorized) incoming request is intercepted by the TokenHider. The access token is removed from the incoming request and stored in a temporary storage of the TokenHider. Then, the access token is replaced with a

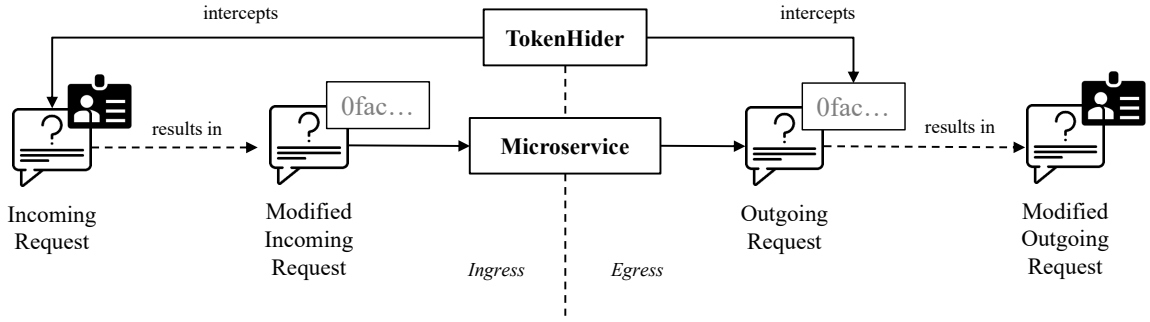


**Figure 6.5:** Overview of the TokenHider

unique identifier (e.g., UUID) that identifies the request. The request is subsequently forwarded to the microservice. If the microservice performs a S2S request and uses identity propagation, instead of the access token, the microservice forwards the unique identifier. However, the target microservice receiving the identifier does not know how to handle this identifier. Therefore, the source microservice communicates with the target microservice through an egress. The egress is provided by the API proxy. If the API proxy receives the request, the TokenHider replaces the unique identifier with the original access token. This allows to remove the access tokens completely from the microservice while allowing the propagation of an access token. The applicability of this approach was also demonstrated by Meadows et al. [MH+23] who show a similar concept which exchange an identifier when passing a request through an outgoing egress.

A more detailed UML sequence diagram for the use case *Remove Car from Fleet* is presented in Figure 6.6. The Fleet Manager performs the gRPC request *removeCarFromFleet* to the microservice FleetManagement. The request is intercepted by the PEP-FleetManagement, which forwards the request to the PDP and enforces the result. To simplify the illustration, these steps have been omitted. Assuming the request is allowed, the request is forwarded to the TokenHider. When using Envoy as an API proxy, the PEP and TokenHider can be implemented as Envoy filters that are applied one after the other [EP-Doc]. In Envoy, each HTTP request has a unique *contextID*. The TokenHider stores the access token in a key-value store and replaces the HTTP header containing the access token with the *contextID*. The request is forwarded to the microservice which performs S2S requests, e.g., removeRentableCar. If the TokenHider receives the S2S request (through an egress), the *contextID* is replaced with the access token and the request forwarded to the PDP-RentalManagement and microservice RentalManagement which processes the request and returns the result to the microservice FleetManagement. Finally, the microservice FleetManagement returns the result to the FleetManager. This marks the closing of the initial HTTP connection. Consequently, the TokenHider removes the access token from its internal key-value storage and the microservice never had access to the access token.

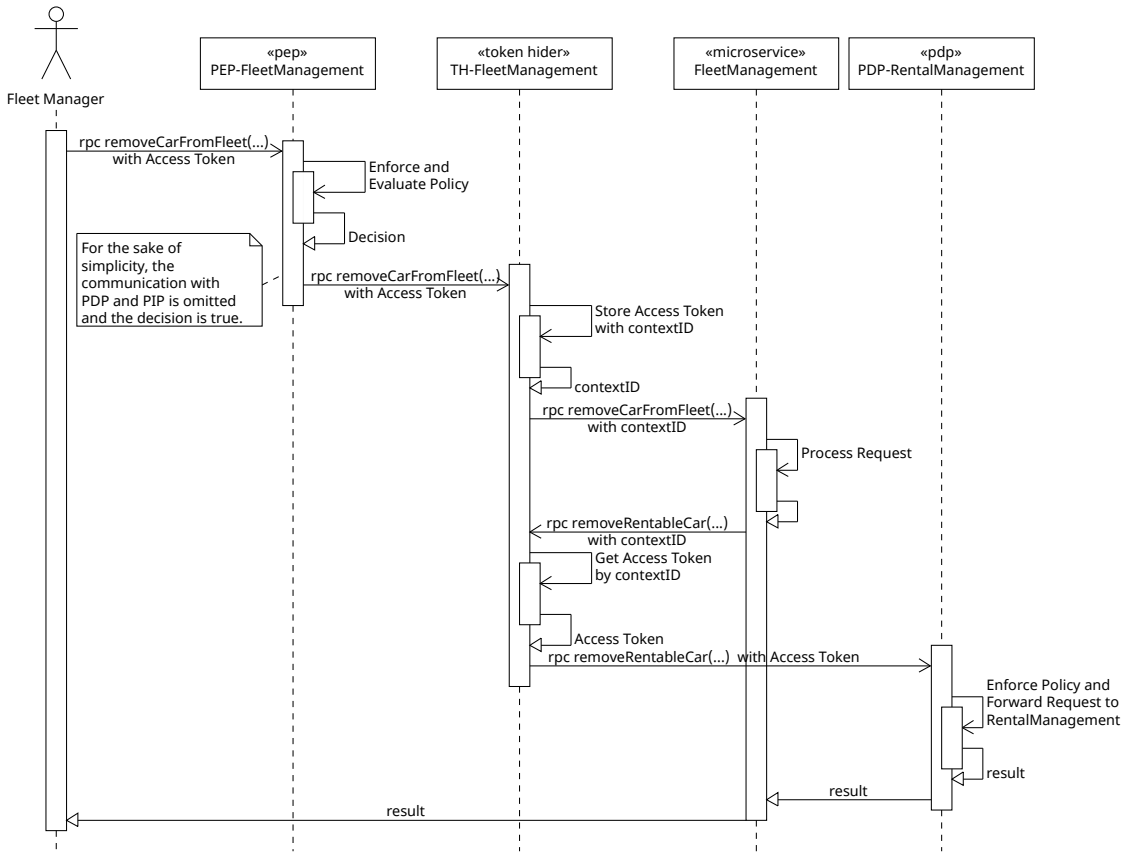


**Figure 6.6:** Sequence to Replace Authorization Tokens

## 6.5 Service-to-Service Requests Resulting From Design Decisions

This chapter elaborated on the S2S requests that are the result of a subject (e.g., human user) interacting with a microservice-based application. Of course, it is also possible that a S2S request is the result of a microservice, i.e., without involving a human user. For example, the frequent (e.g., every 24h) synchronization of data. These requests are the result of the design decisions made for a microservice-based application. Therefore, there is no authorization requirement that can be derived from a functional requirement (e.g., use case description). With the analysis of the software architecture, these requests can be identified with the help of orchestration definitions. With the knowledge of these S2S requests, the authorization policy can be derived and subsequently implemented.

With the lack of an authorization requirement, the granularity of these authorization policies is limited. An example policy is presented in Listing 6.8. The policy only evaluates the action and the object. In addition, the source microservice is evaluated. Additional conditions may be added to the authorization policies depending on the respective design decisions. For example, if the rentable

```
#fleetmanagement_sync_rentable_cars.rego
allow if {
    action_is_sync_rentable_cars
    object_is_rentable_cars
    source_is_fleetmanagement
}
```

**Listing 6.8:** S2S Authorization Policy Without Authorization Requirement

cars should only be synchronized during nighttime, an additional condition evaluating the time can be added. Due to the absence of additional attributes and conditions, the granularity of these S2S authorization policies can be compared to the authorization policies created by Log2Policy [XZ+23] or AutoArmor [LC+21] presented in Chapter 3.

## 6.6 Summary

This chapter introduces the systematic derivation of S2S authorization policies. The approach complements the development of authorization policies presented in Chapter 5 by using the authorization requirements as a foundation. The process begins with the identification of a S2S call in the design phase. To create fine-grained authorization policies, only the S2S requests that are the result of a user interaction with the microservice-based application are considered. By analyzing the software architecture and the respective orchestration definitions, the service interaction collection is created. The collection holds an entry for every S2S request occurring in the application. For each entry, a S2S authorization policy is created. The authorization policies are systematically implemented in the policy language Rego. A particular feature of S2S authorization is the identification (i.e., authentication) of the source microservice starting the S2S request. Therefore, an example using API keys or mTLS is presented. In addition, the identity of the user initiating the S2S calls must be propagated through the microservice-based application to provide fine-grained S2S authorization. This allows to identify who is responsible for initiating the S2S request. To support identity propagation, the microservice must be adapted. Therefore, an example realization requiring minimal code changes is presented on a Golang microservice. Finally, the TokenHider is introduced as a component that allows to remove the access token from the microservice. This reduces the overall attack surface introduced by the identity propagation required for S2S authorization.

# 7 Authorization Application Integration

The previous chapters introduced the development of authorization policies from two points of view: First, the authorization policies that restrict users from accessing a microservice-based application (presented in contribution C1 in Chapter 5). Second, the authorization policies that limit the Service-to-Service (S2S) communication between the microservices of an application (introduced by contribution C2 in Chapter 6). In this chapter, we provide a systematic approach of enforcing the previously implemented authorization policies in a microservice-based application. The integration of authorization into the application requires the creation of artifacts before and after the implementation of the authorization policies. The process introduced in this chapter is the final aspect to address the requirement *R1 - Embedding Authorization Into Development*.

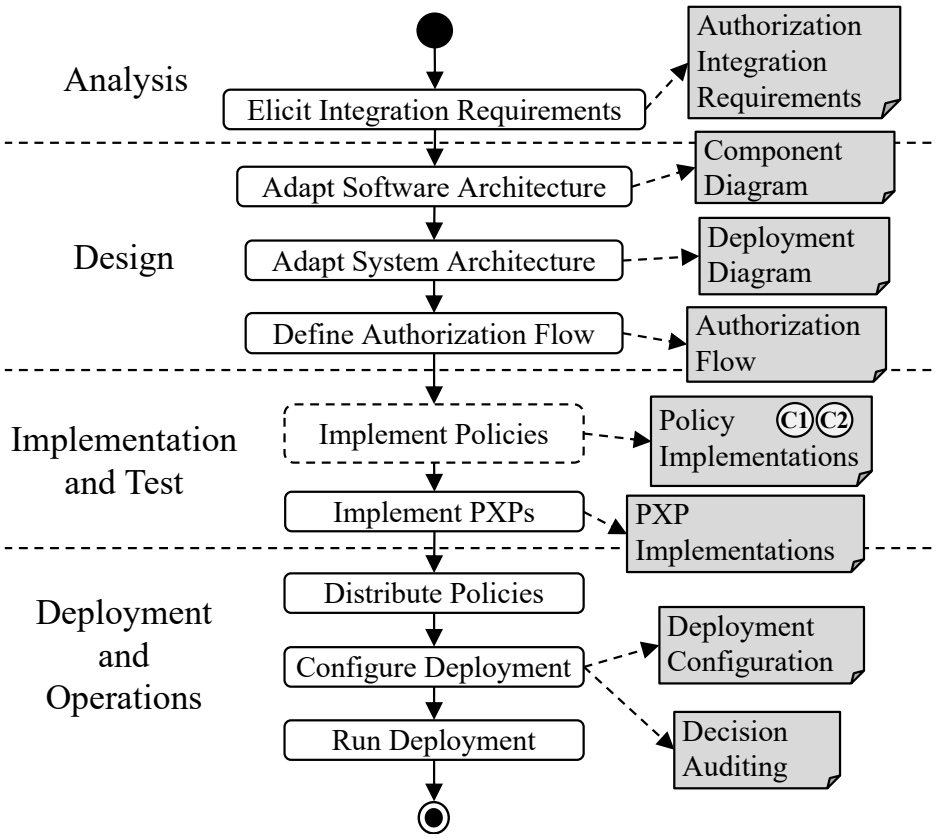


**Figure 7.1:** Systematic Authorization Integration into a Microservice-Based Application

Figure 7.1 depicts an overview of the steps required for the systematic integration of authorization

into microservice-based applications. The process is structured in the phases of analysis, design, implementation and test, and deployment and operations. The remainder of this chapter is structured among the development phases and their required activities. In the analysis phase introduced in Section 7.1, the foundations for the integration of authorization are created. This includes the elicitation of a set of authorization integration requirements which define how authorization should be enforced and which authorization technologies ought to be used. Moving into the design phase introduced in Section 7.2, the authorization integration requirements are realized by adapting the software and system architecture. The software architecture dictates the logical placement of the authorization components required to realize the requirements *R5 - Externalized Authorization* and *R6 - Decentralized Authorization*. By adapting the system architecture, the technology from the authorization integration requirements is introduced to the design through the deployment diagram. This also influences the authorization flow.

The deployment diagram constitutes the transition to the implementation and test phase, which is introduced in Section 7.3. To enforce the Attribute Based Access Control (ABAC) authorization policy implementations, the Policy Decision Point (PDP), Policy Enforcement Point (PEP), and Policy Information Point (PIP) (together referenced as PXP) introduced by the eXtensible Access Control Markup Language (XACML) reference architecture [OAS-XAC] must be implemented. Depending on the required technologies, existing solutions can be used or must be modified.

The final step for the authorization integration into the microservice-based application is the realization of the deployment diagram. This is introduced in the deployment and operations phase in Section 7.4. To support the configuration and the deployment, we propose the use of templates for selected technologies (e.g., Kubernetes). This includes a brief introduction of auditing authorization decisions, as an important operational aspect. In addition, a mechanism to distribute authorization policies to a PDP is provided. A summary of the authorization integration into a microservice-based application is provided in Section 7.5.

## 7.1 Analysis

The authorization requirements created in Chapter 5 are independent of the employed technologies. Additionally, the design of the (S2S) authorization policies in Chapter 5 and Chapter 6 are both independent of a specific technology. However, in the implementation and test phase, the policies are implemented in a policy language, which can also have an impact on the required components in the software architecture. Therefore, in this section, the requirements for the integration of authorization are elicited. Depending on the organizational environment which employs the MAF, the requirements can be adapted.

### 7.1.1 Elicit Authorization Integration Requirements

The authorization integration requirements are a list of non-functional requirements that influence how the authorization is realized in a microservice-based application. Listing 7.1 shows the result of the elicitation. The requirements (*AuthZIntReq*) are numbered and categorized into general, technological, and S2S authorization integration requirements.

```
1  # General
2  AuthZIntReq-10 : The system shall provide fine-grained authorization using
       Attribute Based Access Control.
3  AuthZIntReq-11 : The system shall separate business logic from
       authorization logic by externalizing authorization.
4  AuthZIntReq-12 : The system shall perform authorization decentralized.
5  # Technological
6  AuthZIntReq-20: The system shall use Rego as a policy language to
       implement authorization policies.
7  AuthZIntReq-21: The system shall use \gls{OPA} as a PDP.
8  AuthZIntReq-22: The system shall use Envoy as a PEP.
9  AuthZIntReq-23: The system shall use \gls{OPA} as a PIP.
10 # Service-to-Service
11 AuthZIntReq-30: When the system communicates internally, the system shall
       authorize every request using {coarse - grained, fine - grained }
       authorization policies.
12 AuthZIntReq-31: When the system communicates internally, the system shall
       use { mTLS ( mutual TLS), API tokens, self - signed JWTs } to
       authenticate its subsystems.
13 AuthZIntReq-32: When the system communicates internally, the system shall
       support  identity propagation.
```

**Listing 7.1:** Authorization Integration Requirements

**General Authorization Integration Requirements** The general requirements structure the overall scope of the integration. Since the authorization policies use ABAC, which is also a premise for this thesis (see Section 1.5), the authorization with ABAC is the first requirement (line 2). As presented in Section 3.1, the aspect of externalized authorization (R5) and decentralized authorization (R6) are requirements towards Microservice Authorization Framework (MAF). Therefore, these requirements are considered in lines 3 and 4 respectively.

**Technological Requirements** The goal is to define the used technologies that can have an impact on the architecture or the deployment. The first requirement is the selection of the policy language in line 6. In this thesis, Rego is used as a policy language. This also has an impact on the

PDP that can be used. For instance, Open Policy Agent (OPA) must be used when employing Rego as a policy language (line 7). Next, the PEP must be selected. In this chapter, Envoy is used as an API proxy (line 8) [EP-Doc]. However, this can be changed to another API proxy (e.g., Istio) as long as externalized authorization with the selected PDP (i.e., OPA) is supported. Finally, the PIP must be selected. To access attributes, this chapter uses OPA to access to the microservices backing service (line 9). Similar to the PEP and PDP, the PIP can also be integrated using a dedicated component to access the respective backing service.

**Service-to-Service Requirements** There are various sets of best practices and guidelines regarding the security aspects of employing microservices and the microservice architecture. Based on an analysis of these guidelines with a focus on S2S authorization, we elicit a set of requirements. The Application Security Verification Standard (ASVS) provided by the Open Web Application Security Project (OWASP) foundation [OW21a], the special National Institute of Standards and Technology (NIST) publication regarding security strategies for microservice-based applications [Ch19], and the security standards provided by the UK government [DWP23] are considered. This results in three requirements: First, the internal system communication must be authorized using authorization policies (line 11). In addition, the internal communication requires authentication between the subsystems (line 12). Finally, as described in Chapter 6, the system must support identity propagation to support fine-grained S2S authorization (line 13).

## 7.2 Design

The elicited integration requirements are subsequently realized in the design phase introduced in this section. The starting point for this is the XACML reference architecture (see Section 2.5.2). This includes the adjustment of the software architecture of the microservice-based application by including the authorization components. In addition, the deployment of the microservice-based application including the required authorization components is defined. To orchestrate how the authorization is performed among the introduced components, the authorization flow is specified.

### 7.2.1 Adapt Software Architecture

The MAF employs ABAC to perform authorization decisions with the use of authorization policies. The components of ABAC reference architecture are the PEP, PDP, PIP, and Policy Administration Point (PAP). The goal of the authorization integration is to include authorization per microservice. This removes the security smell of a centralized authorization service [PS+22]. In addition, the authorization components should be externalized from the microservice. This allows to exchange the

authorization logic (i.e., authorization policies) without modifying the microservice, which leads to a higher reusability and maintainability of the microservice.

Figure 7.2 presents an excerpt from the software architecture of the CarRentalApp. The software architecture is depicted by a UML component diagram and presents the software architecture from a logical viewpoint. The UI-CarRental component accesses the business logic of the microservice FleetManagement through an API request. To externalize the authorization logic, for every microservice the components PEP, PDP, and PIP are added to the software architecture. As presented in Figure 7.2, the microservice FleetManagement with its database DB-Fleets is extended with the respective authorization components. The stereotypes presented in Section 4.2.4 are used to model the authorization components.

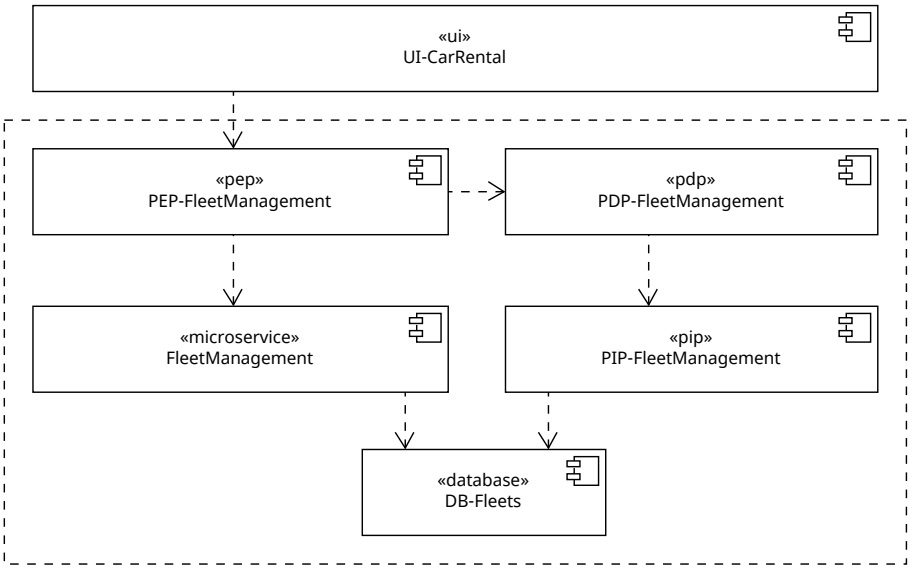


**Figure 7.2:** Placement of Logical Authorization Components in Software Architecture

Notably, the PEP-FleetManagement intercepts the request from the UI and forwards the request to the PDP-FleetManagement to perform an authorization decision. Thus, the authorization decision is not performed in the microservice. To provide the PDP-FleetManagement with the necessary attributes, the PIP-FleetManagement has direct access to the backing service, i.e., database. Since microservices such as FleetManagement should be stateless [FF+18; Wi12], the state is persisted in a backing service such as DB-Fleets. The direct access from the PIP to the respective backing service is a design decision. This decision provides the PIP the most current attributes. Other options include the access from the PIP to an API endpoint of the microservice to access attribute data (1) or a mechanism to push modified attributes from a microservice to a PIP (2). These options require modifications of the microservice, which can reduce the reusability of the microservice in other applications. In addition, requests are added to the authorization flow, which leads to further latency

in the later implementation. In a scalable cloud environment, option 2 also leads to complexity in the persistence of data as multiple states must be managed, requiring additional decisions to address the Consistency Availability Partition tolerance (CAP) theorem [GL02].

The proposed solution with direct access from the PIP to the backing service reduces complexity while providing the most recent attributes. However, it has to be noted that it is still possible to have inconsistent states. This is due to the distributed nature of the application. For instance, depending on the database replication, an updated attribute might not be immediately synchronized with other database instances, allowing unauthorized access for a brief time. In addition, it is important to provide the PIP only with limited access to the database. For example, the PIP only requires the attribute fleetmanager from the DB-Fleets. Thus, the access to the database must be configured accordingly. The PDP should have read-only access to the database and not be able to modify any attributes. The configuration depends on the used database technology. An example for PostgreSQL is further elaborated in Section 7.2.2.

The components depicted in Figure 7.2 can be used to provide the business logic of the microservice with authorization. However, the ABAC reference architecture also includes the PAP which supports the creation of authorization policies and stores the created policies in a repository [HF+14]. The PDP can then retrieve the authorization policies relevant for the microservice (e.g., FleetManagement). This mechanism allows to frequently retrieve the newest authorization policies. Since the policies introduced in Chapter 5 are written in Rego as simple code artifacts, the PAP can be realized by a text editor such as Visual Studio Code. Thus, the PAP is not further elaborated on in the remaining chapter. However, the storage and versioning of the authorization policies in a policy repository is an important aspect. The implemented policies should be stored in a Git repository to introduce versioning. The proposed process for the distribution of authorization policies is further elaborated in Section 7.4.1.

### 7.2.2 Adapt System Architecture

To realize the authorization components depicted in Section 7.2.1, the technologies must be selected accordingly. The authorization technologies are defined in the list of non-functional authorization requirements in Listing 7.1. This has an impact on the deployment of the microservice, including the resource consumption and the scalability. Figure 7.3 depicts an UML deployment diagram for FleetManagement. The deployment diagram moves from the logical view of the overall application presented in the component diagram to a physical view. Envoy is selected to realize the PEP component. Envoy is a lightweight open-source edge and service proxy which is hosted by the Cloud Native Computing Foundation (CNCF) [EP-Doc]. To access the microservice FleetManagement, Envoy provides the only entry point. Since the policies in Chapter 5 are implemented in Rego, OPA is used as a PDP. Instead of introducing a dedicated component realizing the PIP, OPA is extended to allow a

connection to the database DB-Fleets. Thereby, OPA is acting as a PDP and PIP. The extension of OPA is further described in Section 7.3.1.

Envoy provides an *ExtAuthZ* filter which can be configured to forward the request to OPA [EP-EA]. The communication required for the forwarding mechanism between Envoy and OPA is done through gRPC. Therefore, Envoy defines an API which specifies how the messages between Envoy and OPA are parsed [EP-EA]. Other products such as Casbin [CO-Doc] or PlainID [He21] also implement the interface, allowing to exchange OPA.

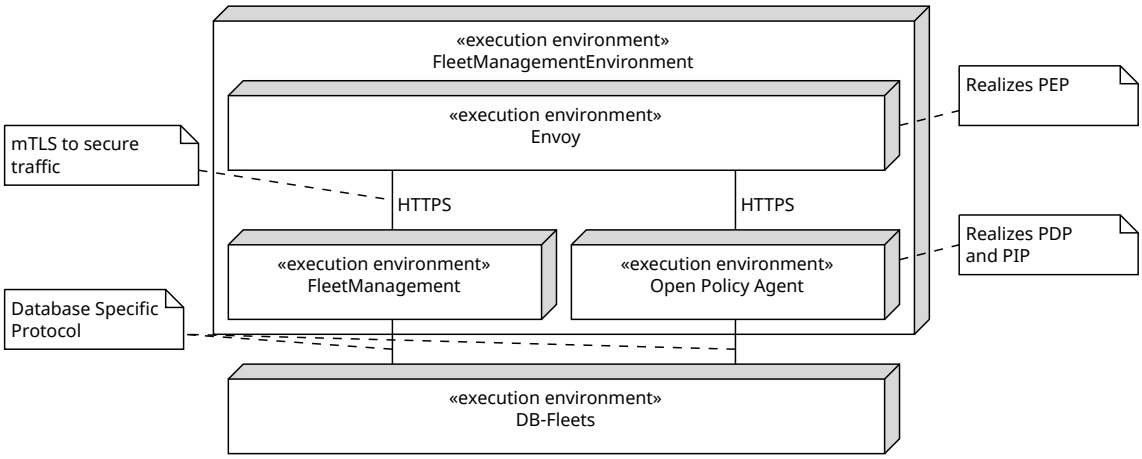


**Figure 7.3:** Deployment Diagram with Proposed Technology Selection

The deployed execution environments Envoy, FleetManagement, and OPA are run in a dedicated execution environment FleetManagementEnvironment. This environment is used to physically combine the authorization components with the microservice as proposed by Miller et al. [MM+21]. If the microservice is to be scaled, the other components in the FleetManagementEnvironment must be scaled up or down respectively. The FleetManagementEnvironment must be configured to only provide an entry point through the Envoy proxy. Otherwise, the microservice could be accessed without providing an authorization mechanism.

To prevent attacks inside the FleetManagementEnvironment (e.g., man in the middle), the traffic should be secured. Otherwise, an attacker gaining access to the environment could intercept the traffic. To secure the communication between Envoy, FleetManagement, and OPA, either HTTPS or mutual TLS (mTLS) can be used [WM17]. Both, Envoy and OPA, support the use of mTLS. The communication between Envoy and the microservice is dictated by the implementation of the microservice. Depending on the deployment, the configuration of the components can be automated to reduce the complexity for a developer (see Section 7.4.2).

### 7.2.3 Define Authorization Flow

The flow of authorization is displayed with an example from the CarRentalApp in Figure 7.4. A request initiated by a subject from the UI-CarRental to the application microservice FleetManagement is intercepted by the PEP-FleetManagement. To evaluate whether the request is authorized or not, the PEP forwards the request to the PDP-FleetManagement. The forwarded request should contain data necessary for authorization, such as the action or the object. Additionally, the token from a subject should be forwarded to provide relevant attributes and check if a subject is authenticated. The PDP-FleetManagement contains the authorization policies for FleetManagement and applies them to the incoming request. Depending on the required attributes, the PDP-FleetManagement requests additional attributes from a PIP, e.g., PIP-FleetManagement. The PIP-FleetManagement accesses the backing service DB-Fleets to retrieve the most recent attributes. Depending on the authorization policy, attributes from multiple PIPs can be requested. The PDP-FleetManagement evaluates the policies and returns an authorization decision to the PEP-FleetManagement. If the decision is allowed, the request is forwarded to FleetManagement which will subsequently process the request and return a result. The result is then returned to the subject. If the decision is denied, an error (e.g., HTTP status code 403) is returned to the subject.

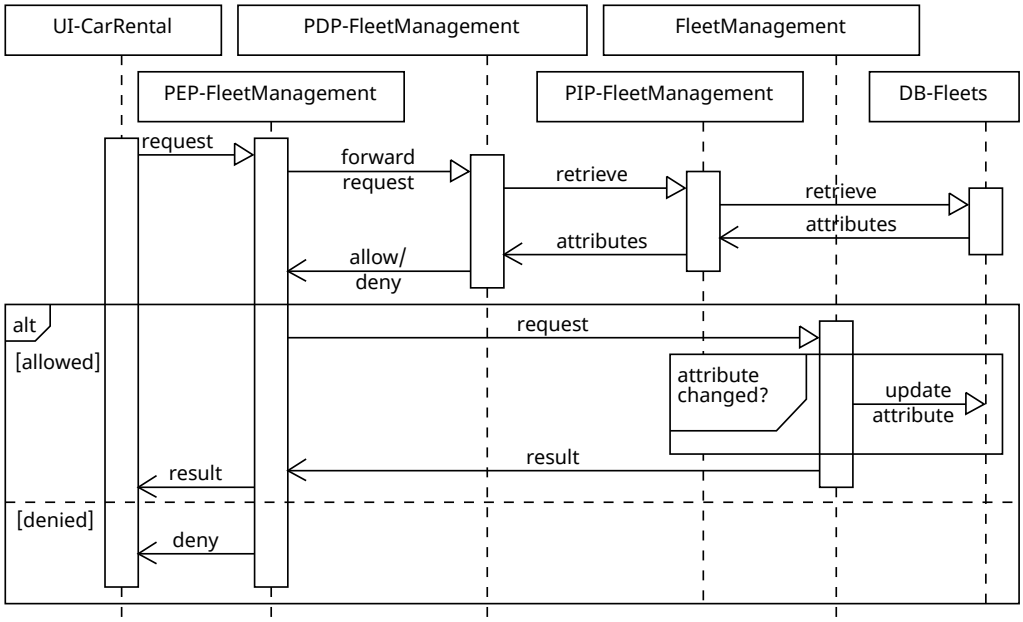


**Figure 7.4:** Authorization Flow for Microservice FleetManagement

When performing a request to a microservice, the microservice might modify the backing service (i.e., database) while processing the request. This also applies to attributes relevant for authorization which are updated. Since the PIP-FleetManagement has direct access to the backing service, the updated

attribute has an immediate effect on upcoming requests. This is especially important in a cloud environment in which multiple instances of a microservice run at the same time. The responsibility of storing and providing the most recent attributes is shifted to the employed backing service, maintaining separation of concerns.

When realizing the logical authorization components, the authorization sequence depicted in Figure 7.4 must be maintained. However, the introduced latency should be considered when selecting technologies (see next section). For example, every request to a PIP introduces latency. Depending on the frequency of attribute changes, caching attributes at the PDP or PIP can be a feasible strategy to reduce the overall latency (see future work in Section 9.2).

## 7.3 Implementation and Test

In the implementation and test phase, the authorization policies introduced in Chapter 5 and Chapter 6 are implemented in a policy language, e.g., Rego. Templates can be used to help a developer implement authorization policies for a microservice-based application. This allows to enforce common guidelines and best practices and to reduce the overall complexity for developers.

For the implementation of authorization policies in a policy language such as Rego, the template should provide a folder structure and pre-defined policies (e.g., default policy). For instance, the structure for Rego policies presented in Section 5.3 should be provided to developers, ensuring that all developers operate on the same structure. Furthermore, the implementation template can contain policy code that is used for multiple microservices. An example of this is the code required for the verification of access tokens (e.g., JSON Web Token (JWT)). To extract attributes from an access token, the signature must be validated first so that the origin of the access token can be verified. For access tokens created using OpenID Connect (OIDC), this process includes OIDC discovery [SB+23], which is further elaborated in Appendix A.4. Since this process follows a standard, the code will not change and can be reused among microservices. Another aspect of the template should be the tool support required for the local development of authorization policies. This ensures that developers, that might be responsible for different microservices, have the same tool sets.

Besides the implementation of the authorization policies in a policy language, the policy points required for the enforcement of authorization policies must be realized. This applies in particular to the implementation of the PDP and PIPs. An exemplary implementation is presented in the following section.

### 7.3.1 Implement PXPs

The realization of the PEP, PDP, and PIP depends on the respective authorization integration requirements and selected technologies. Generally, existing technologies should be employed to realize the authorization components. However, an inherent challenge of ABAC is the distribution of the respective attributes to the PDPs that require them [SO17; AQ+18]. This includes the topicality of attributes, mechanisms such as attribute caching, performance impacts, and secure access to the attributes. In the context of this thesis, the management of attributes is considered as future work. A simple solution for the access of attributes is presented in this section using OPA.

By default, the open-source version of OPA only provides an interface to communicate with an HTTP resource. This allows to retrieve attributes required for authorization. Thus, OPA delivers an option to act as a PIP out of the box. Since microservices are stateless, the data is stored in a backing service such as a relational database [VS+19]. The microservice FleetManagement uses a PostgreSQL database, which is an open source relational SQL database [PG24]. To allow OPA to access the database, OPA must be extended.

OPA provides a plugin support to extend the OPA runtime and Rego with custom functionality [OP-Do]. The developed extension provides a new Rego function called pip.sql.query, which is depicted in line 1 of Listing 7.2. The function accepts two parameters. First, the name of the database as configured in the respective configuration file. Second, the SQL query. Both parameters must be provided as strings.

```
1  result = pip.sql.query("db-rentals", "SELECT *  FROM rentable_car")
2
3  -- result
4  result = [
5  {
6    "brand": "test",
7    "location": "123",
8    "model": "test1",
9    "vin": "981238123"
10 }
11
12 ----
13 result = pip.sql.query(SELECT EXISTS(SELECT * FROM fleet WHERE fleet_id =
       '1' AND fleet_manager = 'fred.brown@bestrental.com'))
14 result = true/false
```

**Listing 7.2:** Database Query Provided by OPA Extension

Lines 4 to 10 of Listing 7.2 present the result of a SQL query. The result is returned as a JavaScript Object Notation (JSON) object which can be accessed in Rego policies. To access the vin attribute, the term *result[0].vin* has to be used. When performing SQL requests, the SQL query should be written to limit the amount of returned results. For example, to evaluate if a user is a manager of a fleet, a query returning all fleets would create a large overhead in data and traffic. In addition, OPA would have to go through the results, find the correct *fleet_id* and compare the *fleet_manager* attribute with, e.g., the mail address of a user. Instead, SQL allows creating complex queries which can be evaluated quickly by the database and return a simple true or false result. An example can be found in lines 13 and 14 of Listing 7.2.

To configure the OPA extension, the configuration file of OPA must be adapted. An example configuration of the DB-Fleets of FleetManagement is depicted in Listing 7.3. Multiple databases can be configured, to retrieve a vast amount of attributes. The configuration of a database requires two parameters. First, a name for the database (see line 4). This name is used to reference the database connection when using the command *pip.sql.query*. Second, the connection URL to access the database. This URL varies depending on the selected database technology.

```
1 plugins:
2   pip_sql:
3     databases:
4       - name: db-fleets
5         url: "postgresql://postgres:password@localhost/postgres?sslmode=
              disable"
```

**Listing 7.3:** OPA Extension Configuration

To limit the access of the OPA extension (e.g., read access only), the database must be configured accordingly. An example for PostgreSQL is provided in Listing 7.4. First, the database user is created in line 1. The user can later be used in the configuration as presented in Listing 7.3. Line 2 creates a limited view on the database *fleets*, which only allows accessing the column *fleet_id* and *fleet_manager*. Line 3 grants the previously created user to only view the content of the limited view. This ensures that OPA cannot modify the database.

```
1 CREATE ROLE opa_user LOGIN PASSWORD 'secretpassword';
2 CREATE VIEW fleetmanager_fleet AS SELECT fleet_id, fleet_manager FROM
      fleets;
3 GRANT SELECT ON fleetmanager_fleet TO opa_user;
```

**Listing 7.4:** Configuration of the Database

## 7.4 Deployment and Operations

In the deployment and operations phase, the microservice must be deployed with the respective authorization components. We consider the deployment to a Kubernetes cluster, a container orchestration system, in this section. If a PDP is deployed with a microservice to a cloud environment, the PDP must know which policies to enforce. This is done by retrieving the policies from a policy storage (also known as policy repository) [HF+14]. Therefore, MAF proposes a mechanism to distribute and retrieve the authorization policies. In addition, the configuration for the deployment to a Kubernetes is introduced. This includes a mechanism to collect decision logs for auditing purposes. Finally, the deployment is run on a cluster.

### 7.4.1 Distribute Policies

Figure 7.5 introduces a process to distribute authorization policies. The process is triggered by the development of a new microservice or by updating authorization policies from an existing microservice. First, the authorization policies are implemented in Rego as described in Chapter 5 and Chapter 6. Subsequently, the policies are pushed to a Git repository to provide a source of truth for the authorization policy. For a microservice-based application consisting of multiple microservices, the policies can all be stored in a central policy Git repository. Storing the policies in a Git repository allows performing code reviews among developers, as well as introducing approval processes.

If policies are pushed to a Git repository, a dedicated Continuous Integration / Continuous Deployment (CI/CD) pipeline can be used to automate the process of releasing the policies to a policy registry. The pipeline displayed in Figure 7.5 includes four steps: First, the code style of the Rego policies is checked. This guarantees a uniform code structure and can enforce Rego style guides (e.g., use of snake_case) or the guidelines of a development team or organization. Second, analogous to pipelines used to build microservices, the policies must be tested before they can be released. If a test fails, the Rego policies likely contain an error. Hence, the pipeline execution is stopped. Third, if the tests succeed, the Rego policies are bundled into a single file. Similar to source code which is compiled to a (single) binary, the Rego policies are bundled into a single *.tar.gz* archive file. The bundle can be used by an OPA client to load the policies.

To avoid loading the policy bundles manually after each update, which might reduce the service quality (e.g., due to restarts), OPA provides a mechanism to frequently pull from a policy storage for a new policy bundle. Therefore, after the pipeline creates the bundle, an additional step pushes (i.e., uploads) the bundle file to a policy storage. Depending on the organizational requirements, an approval workflow including a policy audit as proposed by Brossard et al. [BG+17] can be integrated into the pipeline to avoid accidentally pushing a policy bundle to a policy storage.

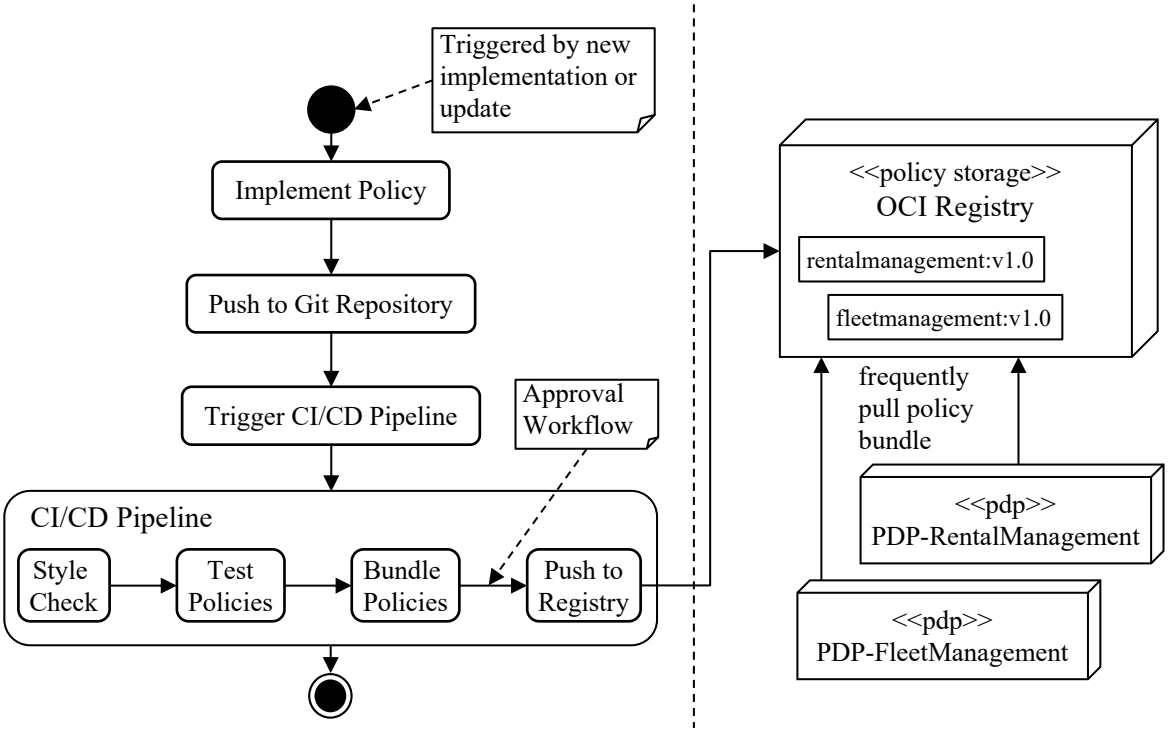


**Figure 7.5:** Distribution of Authorization Policies

There are several options for policy storages. A policy storage can either be a simple web server providing the bundle files or as presented in Figure 7.5 an Open Container Initiative (OCI) registry. OCI introduces a specification for container images which is used for Docker containers [TLF-OCI]. However, this specification can also be used to distribute other artifacts such as Helm charts or OPA policy bundles. Since microservices are typically containerized into Docker images and pushed to an OCI registry, the OPA policy bundles can be placed in the same OCI registry. Finally, the OPA runtime can be configured to frequently pull the required policy bundle. This minimizes downtimes as no restart of the microservice or OPA is required.

To trust the policy bundles and ensure that no malicious actor modified the authorization policies, the policy bundles can be signed (e.g., in the CI/CD pipeline). The signature of the policy bundle can subsequently be verified by the OPA runtime using private/public keys. If the verification of a bundle signature failed, the old bundle will be used. This can guarantee, that only the policies that went through an approval process and were published by the pipeline are used for the microservice-based application.

As introduced before, we propose the use of a Git repository as a source of truth to store the authorization policies of a microservice-based application. Figure 7.6 depicts an example structure for the Git repository storing the authorization policies of CarRentalApp. For each microservice, a dedicated folder is created. Since a microservice can have multiple versions, providing different

functionality, versioning should be considered for the authorization policies. For example, for each version, a folder with the authorization policies for that version can be created. The policy bundle in the OCI registry can then be labeled with the version similar to Docker images (e.g., *rentalmanagement:v1.0*).

```
carrentalapp/
├── FleetManagement/
│   ├── v1.0/
│   └── v1.1/
├── RentalManagement/
│   ├── v1.0/
│   └── v2.0/
└── gitlab-ci.yaml
```

**Figure 7.6:** Structure for Git Repository Containing Rego Policies

### 7.4.2 Configure Deployment

To deploy a microservice with the proposed authorization architecture to a state-of-the art cloud environment, Kubernetes is used. Kubernetes provides an orchestration and management system to run Docker containers [Ku-Doc]. Today, Kubernetes is a commonly used container orchestration platform [Pr24] and is provided by cloud providers such as Google Cloud, Microsoft Azure or Amazon Web Services. Kubernetes deployments are configured declarative through manifest files, which are written in YAML Ain't Markup Language (YAML) or JSON syntax. These manifest files describe a desired deployment state which is to be reached by Kubernetes. However, creating the Kubernetes manifest files can be cumbersome and complex [ZO+23]. Therefore, tools such as Helm exist to support the creation of Kubernetes manifests through so-called Helm charts [CN-AS].

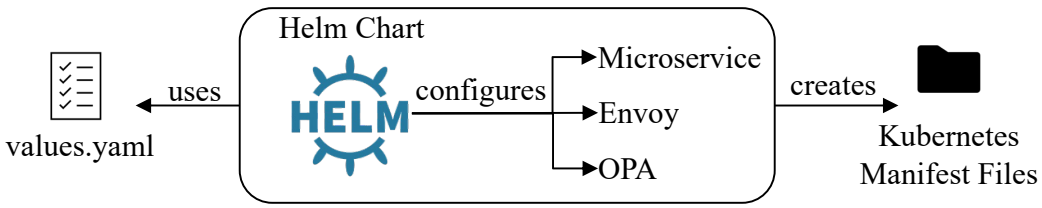


**Figure 7.7:** Configuration Template for Kubernetes Deployment

We propose a Helm chart to reduce the overall configuration complexity of the proposed architecture (see Section 7.2.2) to support developers and to have a common deployment structure inside an organization. Figure 7.7 depicts a high-level overview of the proposed Helm chart. The Helm chart configures the deployment of the microservice container, as well as the Envoy container and the OPA

```
1 opa:
2     policies: <oci-registry>/FleetManagement:v1.0
3     databases:
4         ...
5     decision-service: https://<URL>/..
6     env:
7         ISSUER: ...
8 microservice:
9     env:
10 envoy:
11     ...
```

**Listing 7.5:** Helm Values

container. The Envoy container is configured to be the entry point for incoming requests. In addition, Envoy is configured to forward the requests to the respective OPA and microservice containers. OPA is configured to enforce the implemented authorization policies (see Section 7.4.1) and to log authorization decisions.

A Helm chart typically uses a *values.yaml file*, which configures specific variables for a deployment. An example values can be found in Listing 7.5. For the Helm chart used for the proposed deployment architecture, the location of the authorization policies must be specified (line 2). If OPA should log authorization decisions, a log server must be provided (line 5). In addition, the connection to databases must be defined (line 3) and environment variables required by policies, e.g., for token verification, can be provided. Additional environment variables for the microservice and Envoy can be provided if necessary (lines 8 to 11).

**Auditing Authorization Decisions** Auditing is a complementing aspect of access control [SS94]. To allow an audit of the authorization decisions performed in the proposed architecture, the architecture must be complemented. Figure 7.8 depicts an extension of the deployment architecture presented in Section 7.2.2. A log collector such as FluentD is added to the UML deployment diagram. FluentD is a tool used in a cloud native environment such as Kubernetes which allows collecting logs and forwarding them towards a data sink (e.g., S3 buckets) or a log management tool (e.g., Elasticsearch) [FP-Wha]. The PDP-RentalManagement must be configured to forward the log decisions to a log collector such as FluentD. Depending on the configuration, logs can be forwarded in real-time or in timed batches.

OPA can be configured to forward the decision logs to an arbitrary log collector. The decision logs are forwarded as JSON documents, which must be accepted by a respective log collector. Listing 7.6 depicts a decision log created by a policy evaluation of OPA. The labels in line 1 allow to uniquely identify the PDP and its version which created the decision log. Each authorization decision has a

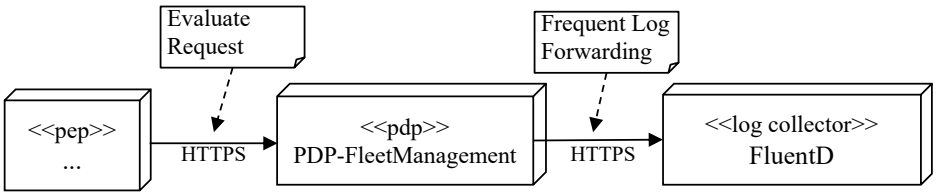


**Figure 7.8:** Distribution of Authorization Decision Logs

unique identifier (line 2). The path of the evaluated authorization policies (line 3), the input provided by Envoy (line 5), and the final result (line 6) are also stored in the decision. Depending on the sensitivity of the data stored in the input, the values of the input can be individually masked. In addition, the timestamp (line 7) is stored with the decision.

```
1 labels = { "id": "UniqueRuntimeID","version": "0.65.0"}
2 decision_id = "3d9d2554-82db-49e1-bff5-c2a0da6d4497"
3 path = "policies/allow"
4 type = "openpolicyagent.org/decision_logs"
5 input = {...}
6 result = true
7 timestamp = "2024-06-20T09:16:50.023446968Z"
8 metrics = { OPAPerformanceMetrics}
```

**Listing 7.6:** OPA Decision Log

Another aspect of the decision log are metrics recorded by OPA (line 8). The metrics include, e.g., the amount of time required to evaluate an authorization policy. Together with the rest of the decision log, in-depth analysis can be performed to evaluate, e.g., the impact of different inputs on the time required to evaluate a policy.

### 7.4.3 Run Deployment

The proposed architecture is deployed to a Kubernetes production environment. As described in Section 7.4.2, the Helm chart is used to configure a Kubernetes deployment. To automate the deployment, a CI/CD pipeline can be used [TH+21]. As depicted in Figure 7.9, the source code of a microservice is stored in a Git repository along the configured Helm chart. The CI/CD pipeline is used to build, test, and deploy the microservice to a Kubernetes cluster. For example, the deployment can be performed through a central repository containing the Helm charts of a microservice-based application also known as an umbrella chart.

If a Kubernetes cluster is used, the required authorization components are deployed to a Kubernetes pod. The pods of a microservice-based application can be managed in a Kubernetes namespace, which

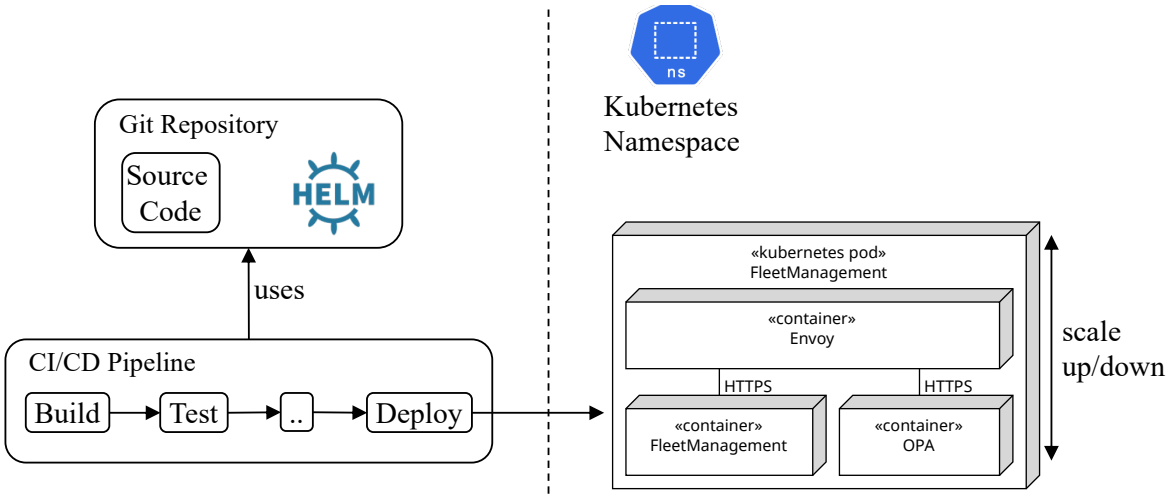


**Figure 7.9:** Deployment to a Kubernetes Cluster

provides a mechanism to isolate resources in a cluster. A pod is the smallest deployable unit which can be managed by Kubernetes [Ku-Doc]. To handle different loads to the proposed architecture, Kubernetes can scale pods up and down using auto scalers [TD+21]. This allows to manage the amount of used resources. As presented in Figure 7.9, Envoy, OPA, and the FleetManagement microservice are each deployed as a Docker container inside the FleetManagement pod. This makes use of the sidecar pattern, which allows multiple containers inside a pod [BO16]. Thus, if the pod is scaled up or down, each container is replicated accordingly, realizing the proposed deployment architecture. In addition, the containers are physically placed close to each other (i.e., on the same Kubernetes node). This supports the reduction of latency generated by the additional communication that is introduced by authorization components.

## 7.5 Summary

This chapter introduces the integration of authorization into a microservice-based application, complementing the creation and implementation of (S2S) authorization policies. The analysis phase elicits the requirements towards the integration, including general requirements, technological requirements, and requirements regarding the S2S authorization. These requirements are subsequently realized in the design phase, which specifies how the components are placed in the software architecture and in a deployment environment and the authorization flow. The reference architecture by XACML is considered here. The architecture externalizes the authorization enforcement. In addition, the architecture does not contain a centralized authorization service. Instead, the PDP is deployed per microservice, decentralizing the authorization mechanism. This also supports the overall scalability of the microservice, demonstrated in the deployment and operations phase using a Kubernetes example.

Finally, an approach to distribute the authorization policies to the PDPs is presented. It allows treating policies as code artifacts and applying development concepts such as CI/CD. The management of attributes remains an open topic and constitutes future work. This chapter proposes a first solution using an extension to OPA. The extension allows accessing attributes by querying a microservice's backing service. However, the approach is currently limited to PostgreSQL databases.

# 8 Validation of the Contributions

In this thesis, we propose the Microservice Authorization Framework (MAF) to support a systematic integration of policy-driven authorization in microservice-based applications. The framework includes three contributions: First, the systematic engineering of authorization policies (C1), which includes the elicitation of authorization requirements and the implementation of authorization policies. Second, the development of Service-to-Service (S2S) authorization policies (C2). Third, the systematic integration of authorization components into a microservice-based application (C3).

To validate the proposed MAF, we follow an empirical validation approach [Sc24; Du16], using the CarRentalApp introduced in Section 1.1 as our exemplary application. The validation approach is divided into several steps and presented in this chapter.

Section 8.1 introduces the empirical validation in the context of software engineering, which is applied in this thesis. The considered validation includes four types of validation, of which three are applied to the MAF. In the subsequent sections, the validation of the respective types is described. In Section 8.2, the feasibility (Type 0) of the MAF is discussed based on the requirements catalog introduced in Section 3.1. Section 8.3 discusses the suitability (Type 1) of the MAF. The applicability (Type 2) of the MAF is elaborated in Section 8.4. Finally, Section 8.5 summarizes the validation.

## 8.1 Overview and Conducted Steps of Empirical Validation

Durdik [Du16] and Giessler [Gi18] describe four types of empirical validation. Type 0 describes the general feasibility of the contributions proposed by the MAF. Type 1 targets the suitability based on a consistent example, which is the CarRentalApp. Type 2 discusses the applicability of the presented approach. Type 3 evaluates and compares the expected costs created by the application of MAF with the provided benefits. Figure 8.1 shows an overview of the types of empirical validation in relation to costs and external validity. The cost (y-axis) implies the monetary or human resources which has to be applied to perform the validation. External validity (x-axis) describes the generalizability of the approach outside the laboratory, i.e., in an industrial context [WR+12].

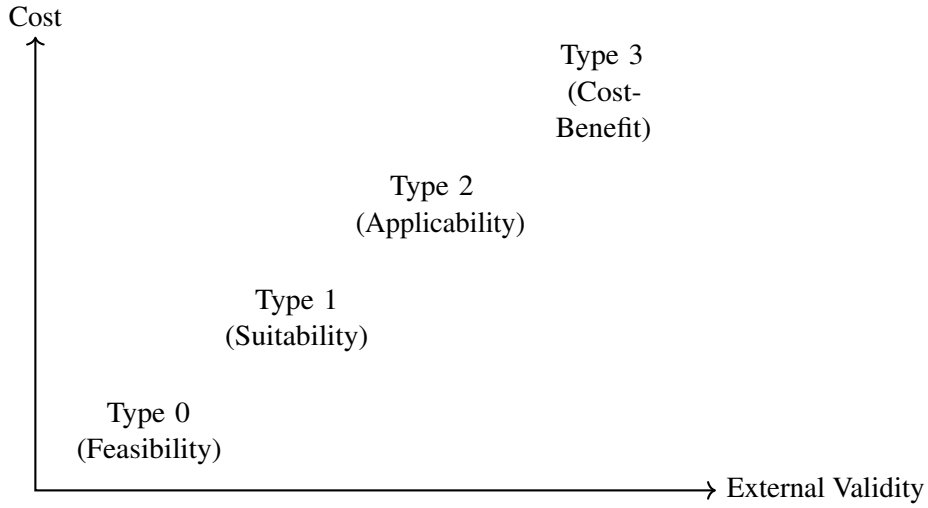


**Figure 8.1:** Types of Empirical Validation in Relation to Cost and External Validity [Gi18]

**Type 0 - Feasibility** The Type 0 validation is the simplest form of validation. The feasibility is evaluated by using fictive examples. Compared to other types of empirical validation, the Type 0 validation can be performed effortlessly at low cost (see Figure 8.1). In this thesis, a comparison is performed between a target state which uses the MAF and a (fictive) present state which does not use the MAF. The present state is captured using the state of the art on policy-driven authorization in microservice-based applications applied to a fictive microservice-based application. The target state uses the systematic integration of authorization proposed by the MAF. The evaluation is performed based on the requirements collected in Section 3.3.

**Type 1 - Suitability** Compared to the feasibility, the suitability of the proposed MAF is evaluated based on a continuous example. The example can still be fictive, as the validation is performed in a lab context [Du16]. However, an example with a strong practical (or industrial) context increases the external validity [Du16]. In the context of this thesis, the CarRentalApp presented in Section 1.1 is used as an example application. The contributions provided by the MAF are applied to the CarRentalApp by the author and are individually evaluated. The evaluation of the proposed MAF is performed based on the observation of the results of the application on the example. Compared to Type 0 validation, the cost of the Type 1 validation is higher, among others, due to the amount of implementation work performed by the author. However, the application on an example implies a higher external validity.

**Type 2 - Applicability** The Type 2 validation is typically implemented in the form of a case study or an experiment. In this thesis, we focus on a case study. The goal of this case study is the evaluation of the applicability of a method, when used by the target users [Du16]. In the case of MAF,

the targeted users are software engineers, software architects, or developers. Ideally, the case studies performed in the Type 2 validation are performed in a field environment using a real-world application scenario, if present. This increases the overall external validity. However, this requires access to a project in an industrial context, which is difficult to obtain. Additionally, this introduces high costs (e.g., working time). An alternative solution, in the absence of real-world applications, is the use of student participants. The use of student participants has been discussed by Tichy [Ti00] and used in case studies performed by Schneider [Sc24] and Durdik [Du16]. In this thesis, we follow the work from Schneider and Durdik, and evaluate MAF with students in the field of computer science. Having a background and expertise in software engineering, this student sample reflects future target users of our framework. The overall effort and thus cost increases due to the case study.

**Type 3 - Cost-Benefit** The Type 3 validation is a cost-benefit analysis introduced by Koziolek [Ko08]. The cost-benefit analysis requires the development of a microservice-based application in a field context (e.g., industrial) using the MAF. To evaluate if the proposed MAF provides a benefit, the Type 3 study must be conducted at least twice. The first case study develops the application without the proposed MAF. The second case study uses the MAF, accepting the potential costs. Comparing the costs for both implementations including the (long term) operational costs enables the assessment of a benefit in cost for the respective business. In addition, the Type 3 study provides a high external validity based on the real-world scenario. Due to the high costs and time required by a Type 3 validation and the lack of an industry scenario, the Type 3 validation is not performed in this thesis.

### 8.1.1 Threats to Validity

Each type of validation comes with limitations and subsequent threats to the trustworthiness and validity of the results. Wohlin et al. [WR+12] list four threats to the validity of the experimentation in software engineering. They differentiate between the validity of the construct, the internal validity, the external validity, and the reliability validity.

**Construct Validity** The construct validity targets the construction of the case study. It represents the extent to which the studied (operational) measures reflect the intentions of the researcher designing the case study.

**Internal Validity** The internal validity targets the quality of the results of the experiments. If the impact of one factor on an investigated factor is researched, there is a risk of a third factor also having

an impact on the investigated factor. The extent to which one or multiple factors have an impact on the investigated factor is a threat to internal validity.

**External Validity** The external validity is concerned with the generalizability of the results of the case study. This targets the extent to which the scenario of the case study can be applied to an industrial scenario.

**Reliability Validity** The reliability validity considers to what extent the results of an case study are dependent on the specific researcher. Ideally, if another researcher performs the experiments, the results should be identical.

### 8.1.2 Goal Question Metric Approach

The Goal Question Metric (GQM) approach has been introduced by Basili et al. [BC+94] and aims at improving the measurement of results in software engineering.

The GQM approach introduces three levels, which are presented in Figure 8.2. First, the conceptual level defines a goal, which is an object of measurement such as products (e.g., artifacts, deliverables), processes (e.g., software related activities), and resources (e.g., items used by processes, hardware). Second, the operational level describes a set of questions used to characterize the assessment of a specified goal. Third, the quantitative level creates a metric which is associated with a question. The data provided by the metric can be objective or subjective. The data is objective if it does not depend on the viewpoint it is taken from. The data is subjective if it depends on the viewpoint from which it is taken (e.g., user satisfaction).

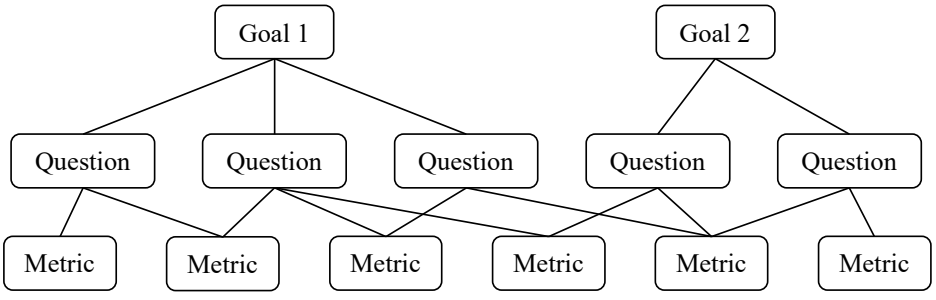


**Figure 8.2:** Hierarchical Structure of the Goal Question Metric Approach According to Basili et al. [BC+94]

Table 8.1 provides an example of the application of the GQM approach. The table is structured using the goal in the first row. The questions and metrics are alternated, located below the goal. The goal is defined by specifying a purpose, an issue, a process, and a viewpoint. Subsequently, the goal is

| | | |
|---|---|---|
| Goal | Purpose<br>Issue<br>Object (Process)<br>Viewpoint | Improve<br>the timeliness of<br>change request processing<br>from the project manager's viewpoint |
| Question | | What is the current change request processing speed? |
| Metrics | | Average cycle time<br>Standard deviation<br>% cases outside of the upper limit |
| Question | | Is the performance of the process improving? |
| Metrics | | $\frac{Current average cycle time}{Baseline average cycle time} * 100$<br>Subjective rating of managers' satisfaction |

**Table 8.1:** Example Application of Goal Quest Metric [BC+94]

refined into a set of questions. Each of the questions can be measured using one or multiple metrics. The same metric can be used for multiple questions.

## 8.2 Type 0 - Feasibility

This section describes the first step of the validation approach, the Type 0 validation. The validation of the feasibility compares the results of the developed MAF with the state of the art presented in Section 3.2. To structure this, we rely on the requirements presented in Section 3.1. In the following, the fulfillment of each requirement is discussed separately. In addition, Table 8.2 visualizes an overview of the assessment. A completely fulfilled requirement is marked by ●. If a requirement is only partly fulfilled, the symbol ◐ is used. A requirement that is not fulfilled by the related work is presented with the symbol ○. If a requirement is not applicable to a publication, the character / is used.

**R1 - Embedding Authorization Into Development** The embedding of authorization into the development of a microservice-based application ensures that the aspect of authorization is considered throughout all phases of the Software Development Life Cycle (SDLC). Authorization should not be considered as an afterthought, e.g., in the implementation and test phase. The contributions of MAF span through the phases of analysis, design, implementation and test, and deployment and operations. Authorization is considered to be an essential aspect of each development phase. The development of authorization policies (C1 and C2) contains new authorization-related artifacts for the analysis, design, and implementation and test phase. The introduced authorization artifacts are systematically created and rely on previously created development or authorization artifacts. The overall integration of authorization into the microservice-based application (C3) is also considered

| | R1 - Embedding Authorization Into Development | R2 - Definition of Authorization Requirements | R3 - Fine-Grained Authorization | R4 - Service-to-Service Authorization | R5 - Externalized Authorization | R6 - Decentralized Authorization |
|---|---|---|---|---|---|---|
| [NJ+18] Fine-Grained Access Control for Microservices | ○ | / | ● | ● | ◐ | ○ |
| [SH+21] ThunQ: A Distributed and Deep Authorization Middleware for Early and Lazy Policy Enforcement in Microservice Applications | ○ | / | ● | ○ | ◐ | ◐ |
| [BG+17] A Systematic Approach to Implementing ABAC | ◐ | ◐ | ● | / | ◐ | ○ |
| [XP+12] Automated Extraction of Security Policies from Natural-language Software Documents | ◐ | ◐ | ○ | / | / | / |
| [XZ+23] Log2Policy: An Approach to Generate Fine-Grained Access Control Rules for Microservices from Scratch | ◐ | / | ◐ | ● | ◐ | ● |
| [LC+21] Automatic Policy Generation for Inter-Service Access Control of Microservices | ◐ | / | ◐ | ● | ◐ | ● |
| **Microservice Authorization Framework** | ● | ● | ● | ● | ● | ● |

**Table 8.2:** Results of the Type 0 Validation

as a task which is performed throughout all development phases. Therefore, the MAF considers authorization holistically throughout the development of a microservice-based application, fulfilling requirement R1.

**R2 - Definition of Authorization Requirements** The creation of authorization requirements forms the foundation for the integration of authorization into an application. An authorization requirement should define who is allowed to perform an operation on a given target and under which circumstances the operation can be performed [Fi03]. MAF introduces the authorization requirement as the central authorization artifact in the analysis phase. The goal of the artifact is the extraction of authorization knowledge, which is contained in the functional requirements. Thus, functional requirements, such as use cases, which define what a user should be able to do with an application are analyzed and the authorization related aspects extracted. This includes the aspects relevant to ABAC, such as the subject, the action, the object, and the conditions. The authorization requirements are formatted into a template, which can be adapted depending on the requirements of the employed development approach. Authorization requirements are the crucial input for the subsequent development of fine-grained authorization policies. Therefore, requirement R2 is considered to be fulfilled by the MAF.

**R3 - Fine-Grained Authorization** Fine-grained authorization allows performing authorization decisions on a granular level. This is required to realize principles such as the principle of least privilege. Access should only be allowed when necessary to perform business logic. Attribute Based Access Control (ABAC), which is employed by MAF, is generally considered to allow fine-grained authorization. To achieve this, ABAC utilizes attributes that stem from the subject, the object, or the environment and allows comparing attributes to create a higher degree of granularity [AQ+18]. However, for the MAF to provide fine-granularity, authorization policies are created. The authorization policies are introduced as an authorization artifact in the design phase. Developers are supported with a systematic approach to create authorization policies. The foundation of the authorization policies are the authorization requirements created during the analysis phase. In addition, Application Programming Interface (API) specifications are used as input for authorization policies, since they define how an object is structured and how it can be accessed. Based on the authorization policies created in the design phase, the policies can be implemented in an policy language. This thesis proposes the use of the Rego policy language, which allows treating a policy as regular code artifacts. By creating fine-grained authorization policies, the requirement R3 is considered to be satisfied by MAF.

**R4 - Service-to-Service Authorization** Moving from a monolithic architecture towards a distributed architecture, such as the microservice architecture, creates communication between the components of the architecture [Ne15]. This is inherently the case in microservice-based applications consisting of multiple microservice. While the single responsibility principle should generally be applied to a microservice, in some cases, communication to other microservices is necessary to provide business logic. With the additional move towards zero trust, these requests should never be

trusted, e.g., due to apparent proximity of the microservices. Instead, S2S requests should always be authorized. Therefore, S2S authorization is an integral part of the MAF (see Chapter 6). For every S2S request occurring after a user interaction, a fine-grained S2S authorization policy is created. To achieve the granularity, the authorization requirements are used. S2S requests that do not depend upon user interaction are briefly addressed, as they heavily depend on the design of the microservice-based application. With the development of S2S authorization policies as part of MAF, requirement R4 is considered to be fulfilled.

**R5 - Externalized Authorization** Externalized authorization allows separating the logic required for authorization from the business logic [HF+14]. This improves the overall flexibility and maintainability of the microservice-based application. If an aspect of authorization changes, the authorization policy can be adapted without having to build, test, and deploy the microservice again. In addition, the logic of a microservice can be re-used in different scenarios using different authorization policies. To provide the externalized authorization, MAF applies the eXtensible Access Control Markup Language (XACML) reference architecture to every microservice of a microservice-based application. This requires an extension of the microservice architecture, which is performed in the design phase and subsequently realized in the implementation and test phase. Every microservice has a Policy Enforcement Point (PEP) (e.g., Envoy), Policy Decision Point (PDP) (e.g., Open Policy Agent), and access to several Policy Information Points (PIPs). This allows to remove the authorization logic completely from a microservice. In addition, the components required for authorization can be quickly scaled up and down when deployed as sidecars to the microservice. With the complete externalization of authorization logic from microservices, the requirement R5 is fulfilled by MAF.

**R6 - Decentralized Authorization** The final requirement is the decentralization of the authorization. Having a centralized authorization service creates a single point of failure in the overall architecture [PS+21]. If the authorization service is unavailable or compromised, the business logic of the application cannot be provided to a user or sensitive data might be leaked. To remove a single point of failure from a (distributed) microservice-based application, MAF decentralizes the authorization mechanism by providing a PDP for every microservice. Since the PDP can be run as a small and lightweight sidecar, the authorization decisions are performed distributed in the microservice-based application. However, the location of the authorization policy storage should be considered. If a single location is used for storing authorization policies that can be retrieved by the PDP, a single point of failure can be created. This must be considered when deploying the authorization extension provided by MAF. Overall, the authorization decisions are performed decentralized, fulfilling requirement R6.

#### Summary

As presented in Table 8.2, MAF fulfills all requirements created in Chapter 3. This is done by employing a holistic view of authorization across the development phases of a microservice-based application. However, the Type 0 validation is a theoretical evaluation of the requirements and relies on the evaluation by the author. This does inherently include a subjective assessment of the requirements. Hence, in the following, a Type 1 validation is performed to demonstrate the suitability.

## 8.3 Type 1 - Suitability

To perform the Type 1 validation, the MAF is applied to two applications. First, the microservice-based application CarRentalApp that has been used as an example throughout this thesis. The CarRentalApp has been developed following the development process proposed by Schneider [Sc24]. Second, an excerpt of the TrainTicket application [ZP+18]. TrainTicket is a microservice-based application that has been developed for benchmarking purposes by Zhou et al. [ZP+18].

The Type 1 validation is structured into six parts. First, Sections 8.3.1 to 8.3.3 apply the MAF contributions to the CarRentalApp. Section 8.3.4 demonstrates a comparison between externalized authorization and internalized authorization in the context of the CarRentalApp. A brief overview of the application MAF to the TrainTicket application is introduced in Section 8.3.5. Finally, the threats to validity following Wohlin et al. [WR+12] are addressed in Section 8.3.6.

### 8.3.1 C1 - Authorization Policy Development

Overall, the CarRentalApp consists of nine use cases which are performed by two actors, a fleet manager and a customer. The use cases are further specified into use cases descriptions following a uniform template [Co00]. These use cases and the respective use case descriptions serve as an input for the derivation of nine authorization requirements. As presented in Section 5.1, this is the optimal case due to the structure of the use cases of CarRentalApp. However, two difficulties occurred during the derivation. First, an ambiguous identification of an object name, e.g., "Customer Rentals". Second, the identification of the relevant conditions. Both of these issues could be addressed in the design phase by analyzing the realization of the respective use cases in the design artifacts (i.e., API specification). The CarRentalApp is developed with a systematic development approach. Therefore, for every authorization requirement, a single API endpoint is created. In the case of CarRentalApp, two microservices FleetManagement and RentalManagement are created. Each microservice has a gRPC API. With the help of the API specifications, the authorization requirements are transformed into authorization policies. To identify the actors fleet manager and customer, Keycloak is introduced

as an open-source Identity and Access Management (IAM) system [KA-Doc]. Customers and fleet managers are registered inside Keycloak and each have a role called *customer* and *fleetManager* respectively. Finally, the authorization policies are implemented in the policy language Rego. To retrieve the required attributes, the PostgreSQL databases of FleetManagement and RentalManagement are accessed by Open Policy Agent (OPA). In total, this results in nine Rego policy implementations, consisting of 20 rules.

### 8.3.2 C2 - Service-to-Service Authorization Policy Development

As presented in Figure 8.3 the application CarRentalApp consists of three microservices. Next to the known microservices FleetManagement and RentalManagement, the microservice Car is part of the CarRentalApp. The microservice Car provides information about cars for a given Vehicle Identification Number (VIN). To provide this information, the microservice Car relies on an external system ES-ConnectedCars. The external system is provided by a third-party service provider (e.g., Car manufacturer). By applying the MAF to the design artifacts of the CarRentalApp, the S2S requests between the microservices are identified. Therefore, all orchestration diagrams are analyzed. In total, there are four S2S requests. The microservice FleetManagement performs two S2S requests to the microservice RentalManagement. These requests occur, if a car is added or removed from a fleet. In addition, the microservices FleetManagement and RentalManagement communicate with the microservice Car if car information is retrieved. However, as mentioned in Section 4.2.2, the S2S requests that are covered by the MAF are limited to the requests that occur between microservices of the microservice-based application. Therefore, the requests between the microservice Car and the external system ConnectedCars are not considered. The microservice Car must therefore follow the authorization specifications of the external service ConnectedCars.

Analogous to the implementation of the authorization policies presented in the previous section, the S2S authorization policies are implemented in the policy language Rego. The implementation of the S2S Rego policies allows reusing existing Rego rules, e.g., *subject_is_fleet_manager*. All microservices of CarRentalApp are implemented in Golang. To support S2S authorization, MAF requires the microservices to support identity propagation. Overall, five lines of code had to be added to the code base of the microservices FleetManagement and RentalManagement to support identity propagation. In addition, API keys are used by the CarRentalApp to authenticate S2S requests. The logic required to handle API keys added three additional lines of code to the microservices FleetManagement and RentalManagement. Although lines of code are not the most significant metric, it indicates that the changes to a microservice required to support S2S authorization are minimal.

### 8.3.3 C3 - Authorization Application Integration

Following the integration of authorization into a microservice-based application results in the software architecture of CarRentalApp as depicted in Figure 8.3. Each microservice has its dedicated authorization components. The microservice Car does not require a PIP because it does not store any data. Instead, the PDP-Car accesses the PIP-FleetManagement to evaluate if a car is part of the fleet of the fleet manager. CarRentalApp uses OpenID Connect (OIDC) provided by the IAM system Keycloak [KA-Doc]. After the subject is authenticated, it receives an access token. The access token is included in all requests performed by the subject. To validate the received access tokens, the PDP-FleetManagement, PDP-RentalManagement, and PDP-Car will perform OIDC discovery [SB+14] and retrieve the required keys (see Appendix A.4). As presented in the previous section, the external system ES-ConnectedCars is outside the control of the CarRentalApp and defines its authorization mechanism. Thus, no authorization components are added to the ES-ConnectedCars.

The CarRentalApp completely externalizes the authorization. Besides the support of identity propagation, the microservices do not contain any logic required for authorization. Instead, the microservices only provide their business logic through an API. The CarRentalApp is deployed inside Docker containers. OPA is used as a PDP. The PEPs are realized through the Envoy API proxy, which are configured to forward requests to the respective OPA instance. The microservices are either deployed using Docker compose [Do-Do] or Kubernetes [Ku-Doc]. In both cases, each microservice is deployed with an instance of the Envoy proxy and OPA. This allows individual scaling and decentralizes the authorization decision in the application. The authorization policies implemented in Rego are managed in a Git repository and directly mounted into the respective OPA Docker containers.

**Impact on Performance** By integrating the authorization components into the CarRentalApp, additional communication is introduced (see Section 7.2.3). This communication is inherent when aiming for externalized and decentralized authorization. To determine the impact of the additional communication on the performance of the microservice-based application, various metrics can be collected [GL+18]. In this section, the response time introduced by the additional communication is examined. Response time has an impact on how the user experiences the interactions with the system [DB13]. A common practice to determine response time is so-called load testing [Me02].

To perform the load tests, the load testing tool k6 by Grafana labs is used [GL-k6]. k6 allows specifying load tests using JavaScript. To evaluate the additional response time, the load tests are applied to four configurations of CarRentalApp. First, as a baseline, the requests are performed to the CarRentalApp without any authorization mechanism. Second, the authorization components are added without S2S authorization. Third, S2S authorization is added to the second environment. Finally, the authorization components are added, including S2S authorization and the TokenHider introduced in Section 6.4. For each configuration, 1000 requests are performed to the microservice

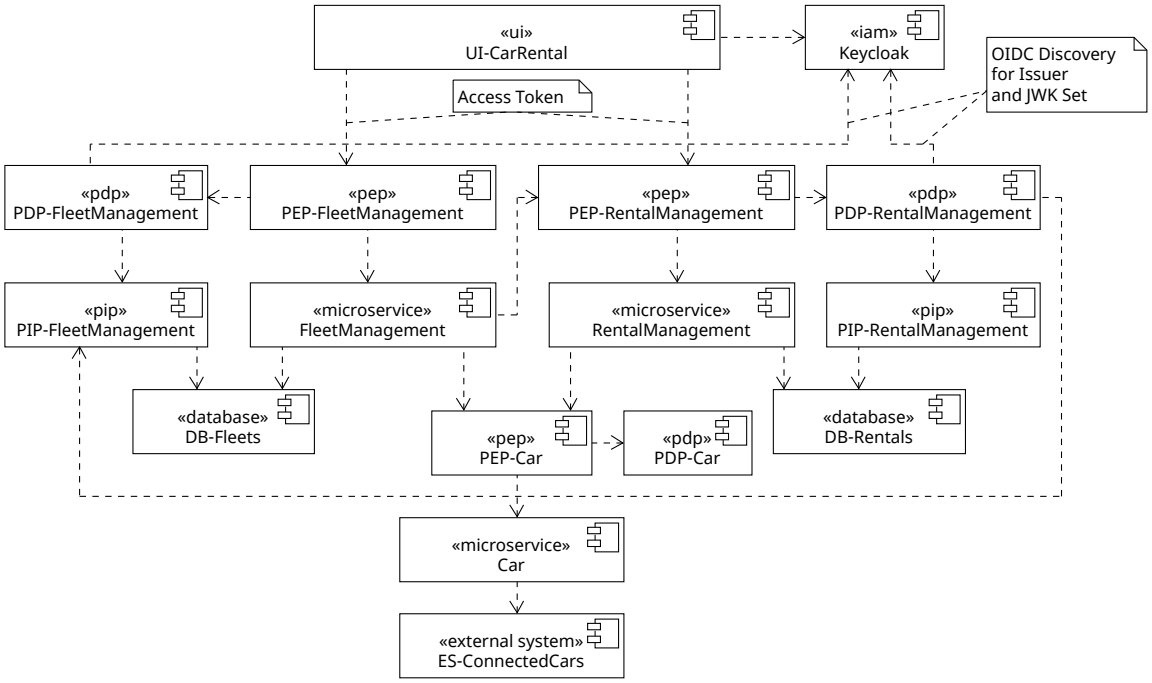


**Figure 8.3:** Software Architecture of CarRentalApp with Authorization Components

FleetManagement. The requests reflect the use cases *List Cars in Fleet* and *Remove Car From Fleet*, which are performed by a fleet manager and are implemented by the RPCs *ListCarsInFleet* and *RemoveCarFromFleet* respectively. The RPC *RemoveCarFromFleet* is a S2S request between the microservice FleetManagement and the microservice RentalManagement.

For the execution of the load tests, the CarRentalApp is deployed using a Docker environment on a desktop machine with an AMD Ryzen 5 3600XT (6 core @ 3.80 GHz) processor with 32 GB RAM. The requests are performed sequentially. Every microservice, including the authorization components, is deployed in a separate Docker container. During the execution of the load tests, the authorization components do not exceed their assigned resources. The results of the load tests are presented in a bar chart including an error bar (standard deviation) in Figure 8.4.

For the RPC *ListCarsInFleet*, the median response time without authorization is 1.5 ms. When including the authorization components, the median response time increases to 3.8 ms. This means an increase of 2.3 ms introduced by the authorization components. The increase in response time can be explained by the additional requests. Compared to the baseline (i.e., direct access), four requests are required in total by the proposed architecture. However, this is inherent when externalizing and decentralizing the authorization mechanism. Since the RPC *ListCarsInFleet* does not include a S2S request, the introduction of S2S and the TokenHider has no effect on the median response time.

Considering the RPC *RemoveCarFromFleet*, the baseline without authorization has a median response

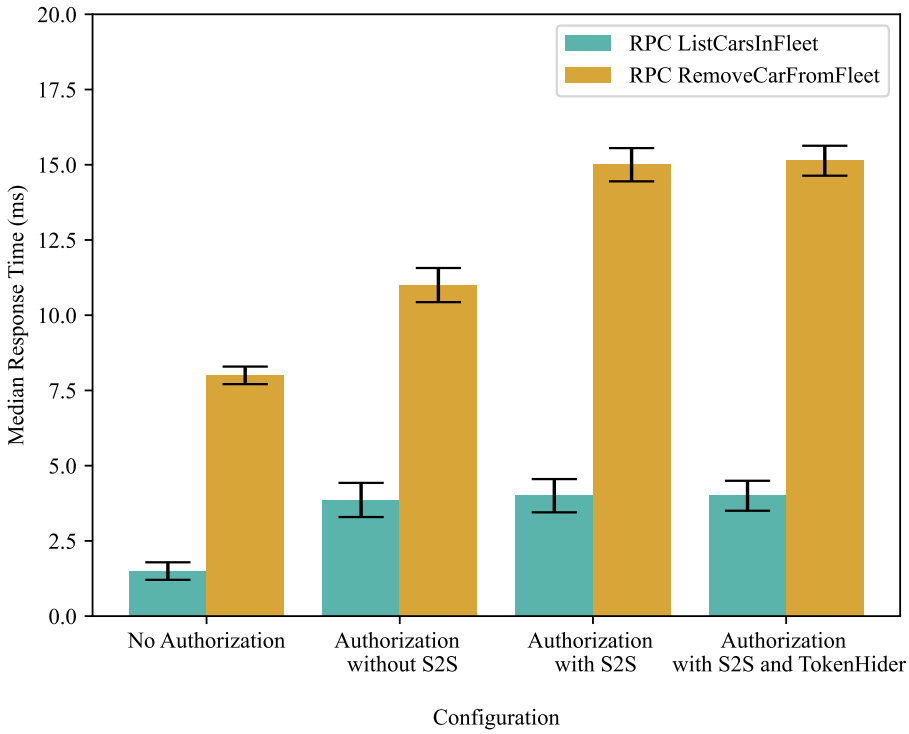


**Figure 8.4:** Median Response Times for 1000 Requests

time of 8 ms. Introducing authorization components without S2S authorization increases the median response time to 11 ms. This increase of 3 ms is consistent to the increase that can be observed for the RPC *ListCarsInFleet*. When introducing S2S authorization, the response time increases to 15 ms. Overall, externalized authorization including S2S authorization introduces 7 ms compared to the baseline. The TokenHider, has a negligible effect on the median response time, with an increase in the median response time to 15.13 ms.

In addition to the response time, the resource consumption of the additional components should be analyzed. This allows to determine the overall cost introduced by the proposed architecture. However, this is considered to be future work (see Section 9.2).

### 8.3.4 Comparison of Externalized Authorization with Internalized Authorization

To compare the externalized authorization introduced by MAF, a version of the CarRentalApp with internalized authorization has been implemented. Therefore, instead of implementing the authorization policies created throughout the design phase in the policy language Rego, the authorization policies are realized inside the code of the respective microservice. Figure 8.5 depicts the implementation architecture of the CarRentalApp's microservices. The implementation architecture is proposed by

Schneider [Sc24] and influenced by the hexagonal architecture [Co05] and clean architecture [Ma12]. The primary structure of the business logic provided by the microservice is presented on the left side. It consists of three parts: First, the API contains the API controller, responsible for managing incoming requests. Second, the logic part implements the business logic (i.e., operations) and contains the model of the microservice. Third, the infrastructure part provides access to backing services (e.g., relation databases such as PostgreSQL). To provide internalized authorization, the implementation targets a loose coupling. Therefore, for each part of the implementation architecture presented on the left side of Figure 8.5, a corresponding part is provided in the internalized authorization extension on the right side. The API part provides the functionality to extract and validate incoming tokens. Furthermore, the extraction of the subject from the access token is provided through the *SubjectExtractor*. The *i-authz_logic* part contains a function deciding the access for every operation in the logic part. In addition, a model defining the subjects (e.g., fleet manager) is introduced. Finally, the *i-authz_infrastructure* part provides the required attribute data to evaluate authorization decisions.

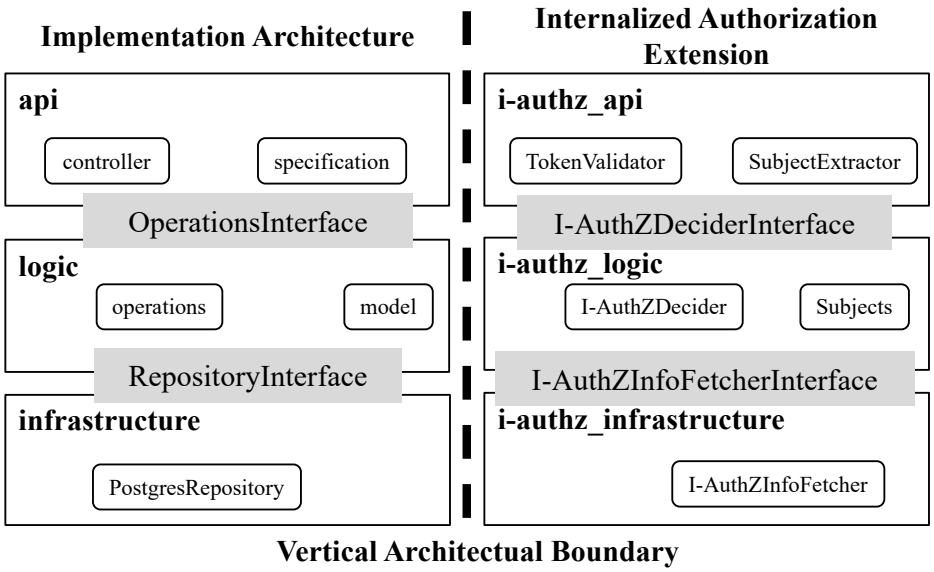


**Figure 8.5:** Implementation Architecture Used for Internalized Authorization

Effectively, each authorization policy results in a so-called decider function in the *i-auth_logic*. Listing 8.1 presents an example of a decider function for the use case *List Cars in Fleet*. The function is called by the *i-authz_api* part, if a request arrives at the API controller. The function receives a *fleetManager* and a *fleetID* as parameters. First, the fleet object is retrieved from the repository (lines 2 to 6). If an error occurs, the request is denied by returning false (line 5). To evaluate if the requesting subject is the fleet manager of the requested fleet, a comparison is performed (line 8) and subsequently returned (line 9). Depending on the outcome of the comparison, the request is either allowed or denied.

```
1 func (auth IAuthZDecider) DecideFleetManagerListCarsInFleet(fleetManager
      subjects.FleetManager, fleetID string) bool {
2   fleet, err := auth.fetcher.GetFleet(fleetId)
3   if err != nil {
4     log.Print(err)
5     return false
6   }
7   // FleetManagerAssignedToFleet
8   fleetManagerAssignedToFleet := fleet.FleetManager == fleetManager.Email
9   return fleetManagerAssignedToFleet
10 }
```

**Listing 8.1:** Function to Decide Use Case "List Cars in Fleet"

By employing the proposed implementation architecture, the authorization policies created by using contributions C1 and C2 can be realized inside the microservices. However, two limitations of implementing the authorization internally are introduced in the following. First, for the microservice to perform fine-grained authorization using the authorization policies created during the design phase, attributes are required. If attributes required for authorization are not managed by the respective microservice, the microservice must retrieve the attributes from another microservice. In addition, the microservice must have an understanding of how the required attributes and entities are structured. This introduces information not necessarily required by the microservice. The attributes managed by another microservice must be retrieved. If an API providing the required information exists, an API request can be performed. This creates additional S2S requests. Otherwise, a new API endpoints must be created or the backing service maintaining the database must be accessed to retrieve the attributes. All options create a coupling between different microservices, which contradicts the overall idea of a loosely coupled microservice architecture.

Second, by implementing the authorization inside the microservice's source code, the maintainability, and reusability of the microservice is reduced. Whenever an aspect of the authorization logic changes, the code of the microservice must be adapted, tested, compiled, and deployed again. In addition, if a microservice were to be used in different applications or settings, the source code must be adapted. This also requires an understanding of the programming language and the implementation structure of the microservice.

Overall, internalizing the authorization allows realizing the authorization policies created during the design phase inside the microservice's source code. While the internalization contains major drawbacks, for smaller applications and development teams, it might be a viable option as it limits the complexity introduced by the deployment (e.g., configuration of components). In addition, if the internalized authorization is introduced loosely coupled into the microservice as presented in Figure 8.5, the effort required to migrate to externalized authorization is limited.

### 8.3.5 Applying MAF to TrainTicket Application

To further examine the suitability of the solutions provided by MAF in an application that is not developed by the author, the microservice-based application TrainTicket [ZP+18] is used as a fully functional, second example application. Therefore, the contributions of MAF are applied to a selected use case of the TrainTicket application and, since the TrainTicket application is a fully working application, the possibility of a retrospective application of MAF is presented.

TrainTicket is a fictional application that provides the functionality to buy train tickets as well as other amenities (e.g., seat reservations, snacks) [ZP+18]. The microservice-based application consists of 41 microservices. The documentation and code are provided through a GitHub repository [FS-Tra]. Most of the microservices are developed using Java. TrainTicket utilizes an internalized authorization mechanism based on Spring Security [Sp-Aut]. In addition, authorization is performed using Role Based Access Control (RBAC) based on roles contained inside a JSON Web Token (JWT).

Unfortunately, the TrainTicket application does not provide any analysis artifacts, describing the functional requirements of the TrainTicket application. The design artifacts are also limited to an abstract overview of the software architecture of the application. Therefore, based on the interaction with the TrainTicket application, a use case has been extracted that includes several S2S requests. The respective analysis and design artifacts have been created for this use case to allow the derivation of the authorization artifacts introduced by MAF. The use case is called *Pay a Reservation Using an External Payment Provider*. It allows a customer of TrainTicket to search for an existing reservation and pay for it by clicking on a button. If the customer has not enough money in the account, the customer is forwarded to a third-party payment provider. The respective use case description can be found in the Appendix in Listing A.6.

Three microservices are involved in performing the respective use case. The microservice *ts-inside-payment-service* receives the request to pay an order from a customer. The microservice will then perform a S2S request to retrieve the order from the microservice *ts-order-service*. If the customer does not have enough money in their account, an additional S2S request is performed to the microservice *ts-payment-service*. After a successful payment, the order is updated, performing an addition S2S request to the *ts-order-service*. An overview of the orchestration definition is presented in the Appendix in Figure A.2. By applying the MAF to create authorization policies, one authorization policy and three S2S authorization policies are created. The authorization artifacts are presented in Appendix A.5.1.

The authorization policies are implemented in the Rego policy language. To apply the architecture proposed in Chapter 7, the selected microservices of TrainTicket do not require code modifications. The internalized authorization mechanism can be left in place. By default, the microservices support identity propagation, which is also used by the internalized authorization mechanism.

To measure the impact on response time of the TrainTicket application, load tests are performed using the load testing tool k6 as presented in Section 8.3.3. Overall, 1000 requests to pay an order are sent to the microservice *ts-inside-payment-service*. Each of these request will result in three S2S requests. Similar to before, five deployment configurations are compared with each other: As a baseline, TrainTicket is used with the included internalized authorization. Second, the baseline is extended by API proxies. Third, the architecture proposed in Chapter 7 is used without a PIP. Fourth, the PIPs are added to the architecture. Fifth, the TokenHider is used. Analogous to the load tests for CarRentalApp, for the execution of the load tests, the TrainTicket is deployed using a Docker environment on a desktop machine with an AMD Ryzen 5 3600XT (6 core @ 3.80 GHz) processor with 32 GB RAM. During the execution of the load tests, the Docker containers did not exceed their allocated resources.

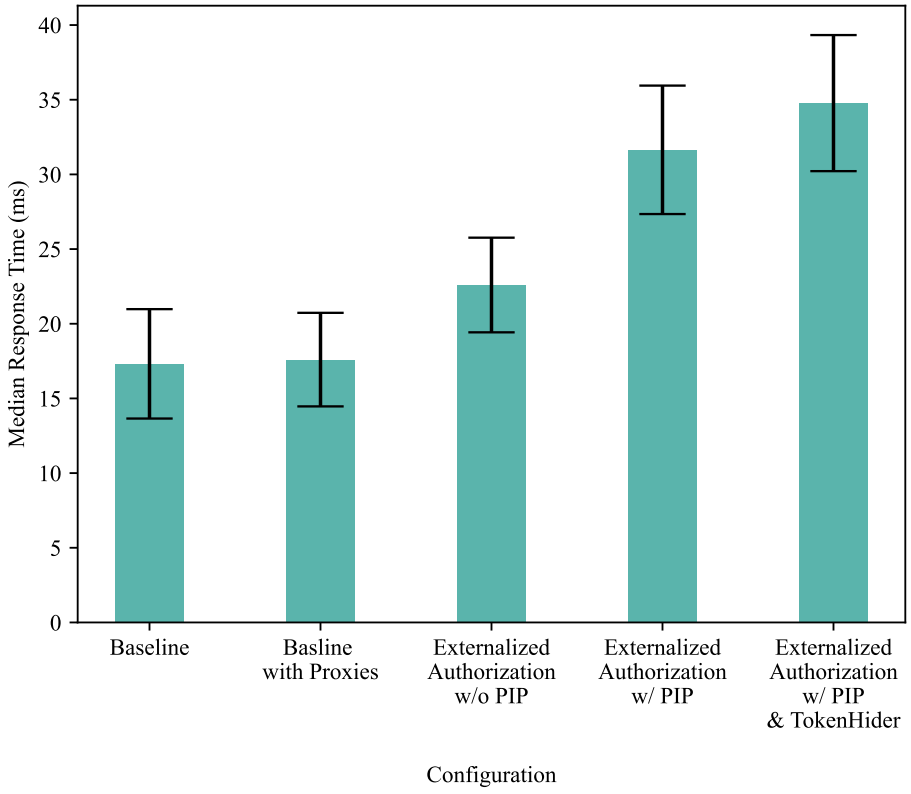


**Figure 8.6:** Median Response Time Impact of Applying MAF to TrainTicket

Figure 8.6 depicts the results of the load tests including an error bar (standard deviation). The baseline has an median response time of about 17 ms. Adding Envoy API proxies to the microservices has a negligible impact on the median response time. When introducing the externalized authorization using OPA without a PIP, the median response time is increased to 22 ms. Introducing a PIP to the deployment leads to an increase of 7 ms to a median of 31 ms. Finally, introducing the TokenHider adds an addition of 3 ms to the median response time. Introducing the PIP to the deployment has the

most significant impact on the response time. For this setup, the microservices act as PIPs themselves and OPA will request the data from the respective microservice (e.g.,*ts-order-service*). This solution is likely to introduce more response time than others, as the TrainTicket microservices using Java are rather slow, as can be seen in the baseline response time when compared to CarRentalApp.

By applying MAF to an existing application, the suitability for the use in existing applications can be shown. However, as can be seen in the example of TrainTicket, the respective analysis and design artifacts must be created to derive the knowledge required to specify authorization policies.

### 8.3.6 Threats to Validity

To put the demonstrated suitability of MAF on the developed CarRentalApp and the external TrainTicket App into perspective, the threats to validity of the suitability are discussed in this section. To structure this, the threats to validity introduced in Section 8.1.1 by Wohlin et al. [WR+12] are used.

**Construct Validity** A first threat to construct validity is introduced by the definition of suitability. While this section showed that the contributions of MAF can be applied to two applications successfully, further metrics beyond measuring success of implementation and response time could be investigated. Another threat is the application of MAF by the author. This includes, for instance, implicit assumptions and additional knowledge by the author.

**Internal Validity** The internal validity can be limited by unaccounted third factors that increase success of the implementation of the contributions to the selected applications. For instance, the design and structure of the use cases can favor the derivation of authorization requirements.

**External Validity** The extent of the external validity is limited by the premises of a structure-preserving engineering approach for the development of a microservice-based application and the definition of structured functional requirements (e.g., use cases). For instance, if there is no documentation in the analysis and design phase such as in the TrainTicket application, developing the authorization artifacts required to implement authorization policies is impossible. In addition, if a transfer between artifacts, e.g., a functional requirement and an authorization requirement, cannot be performed using a one-to-one mapping, the artifacts must be further investigated to extract the required ABAC information (see Section 5.1). However, the internalized implementation of the authorization presented in Section 8.3.4 shows the generalizability of aspects of the contributions C1 and C2 in a microservice-based application using internalized authorization.

**Reliability Validity** The reliability validity can be influenced by the differences in results between developers. An important aspect of the development is the derivation of authorization requirements throughout the analysis phase. Identifying what is relevant for authorization and what is not can be challenging at this early development stage. In addition, there are several degrees of freedom when it comes to the derivation of authorization artifacts (e.g., naming, identification). We aim to limit these degrees of freedom by providing guidelines (e.g., style guide).

### 8.3.7 Summary of Type 1 Validation

The Type 1 validation regards the suitability of the proposed solution. In the case of MAF, the suitability has been shown in Section 8.3 by applying MAF to CarRentalApp and TrainTicket. The primary limitation is the development approach and the respective development artifacts used for the microservice-based application development. When deploying an application using MAF in an production environment, metrics besides the response time should be considered. The greatest threat to validity is posed by the use of the MAF by the author. To address this, the applicability of the MAF is explored subsequently by performing a case study in which developers have to apply MAF to an excerpt of the CarRentalApp.

## 8.4 Type 2 - Applicability

The Type 2 validation aims at evaluating the applicability of the MAF. Therefore, a case study with 15 participants is performed in the context of the development of the microservice-based application CarRentalApp. The case study investigates if the MAF supports developers with the implementation of authorization policies. The definition for a case study by Runeson et al. [PM+12] described in the following is employed for Type 2 validation.

> *Case study in software engineering is an empirical inquiry that draws on multiple sources of evidence to investigate one instance (or a small number of instances) of a contemporary software engineering phenomenon within its real-life context, especially when the boundary between phenomenon and context cannot be clearly specified.* (Runeson et al. [PM+12, p.12])

For the case study, the GQM approach is used as a vehicle to measure and interpret the results. The goals are introduced in Table 8.3 and follow the structure proposed by Basili et al. [BC+94]. The first goal aims at evaluating the applicability of the systematic development of user authorization policies from the viewpoint of a software engineer. This goal targets the contribution C1, which proposes an approach to the development of authorization policies. In contrast, the second goal evaluates

the applicability of the systematic development of S2S authorization policies from the viewpoint of a software engineer. The goal G2 targets the contribution C2, which provides an approach to the systematic development of S2S authorization policies.

The applicability of the contribution C3 regarding the integration of authorization into the microservice architecture is not evaluated because the integration of the authorization components into the microservice-based application can be performed by introducing templates. In an organization using MAF, these templates can be created once and reused for multiple applications, removing repetitive work (see Chapter 7). In the case study, the participants receive a working application which only requires the implementation of the authorization policies.

| **Goal G1** | |
|---|---|
| Purpose | Evaluate |
| Issue | the applicability |
| Object | of the systematic development of authorization policies |
| Viewpoint | from the software engineer's point of view |
| **Goal G2** | |
| Purpose | Evaluate |
| Issue | the applicability |
| Object | of the systematic development of service-to-service authorization policies |
| Viewpoint | from the software engineer's point of view |

**Table 8.3:** Goals for the Case Study

### 8.4.1 Goal Question Metric Plan

A plan for the GQM (see Section 8.1.2) is created to measure the results of the case study. For each goal, a set of questions is created to achieve the goal. As described by Koziolek, for each question the metrics used to answer the question are formulated [Ko08a]. Depending on the question type, either objective collected data (e.g., performance metrics) or subjective data (e.g., user satisfaction on a Likert scale) in the form of questionnaires can be used as metrics.

The subsequent paragraph presents the GQM plan for the goals presented in Table 8.3. For Goal G1, four questions (Q1.1 - Q1.4) are asked. Each question has at least one metric (M). For Goal G2, three questions (Q2.1 - Q2.3) are asked. As with goal G1, each question has at least one corresponding metric. An overview of the GQM plan is depicted in Appendix A.5.2.

**G1** Evaluate the applicability of the systematic development of authorization policies from the software engineer's point of view.

**Q1.1** Can the authorization requirements correctly be derived from analysis artifacts and formulated using the guidelines of the MAF?

**M1.1** % of correctly created authorization requirements.

**Q1.2** Can the authorization policy correctly be created using the guidelines of the MAF?

**M1.2** % of correctly derived authorization policies per authorization requirement.

**Q1.3** Can the authorization policy correctly be implemented using the guidelines of the MAF?

**M1.3** % of correctly implemented authorization policies.

**Q1.4** Does the MAF support the software engineer to create authorization policies?

**M1.4** Perceived ease to create authorization artifacts (for each authorization artifact, Likert scale 1 to 5).

**M1.5** Perceived support provided by the guidelines to create authorization artifacts (for each authorization artifact, Likert scale 1 to 5).

**G2** Evaluate the applicability of the systematic development of S2S authorization policies from the software engineer's point of view.

**Q2.1** Can the S2S requests be correctly identified and subsequently be transferred into authorization policies?

**M2.1** % of correctly detected S2S interactions.

**M2.2** % of correctly created S2S authorization policies.

**Q2.2** Can the S2S authorization policies correctly be implemented using the guidelines of the MAF?

**M2.3** % of correctly implemented S2S policies.

**Q2.3** Does the MAF support the software engineer to create S2S authorization policies?

**M2.4** Perceived ease to create S2S authorization artifacts (Likert scale 1 to 5).

**M2.5** Perceived support provided by the guidelines (Likert scale 1 to 5).

To determine the correctness of the created artifacts, the solution of applying MAF to the CarRentalApp created in the Type 1 validation is selected as reference. If the content of the participants' solution aligns with the sample solution, the solution is considered as correct. This includes minor syntactical errors (e.g., upper or lower case errors). However, if there are differences on a semantic level, such as the identification of the wrong object, the artifacts are considered as false.

### 8.4.2 Case Study

The case study aims at answering the questions introduced by the GQM plan in the previous section. The case study is performed with students attending the courses "Praxis der Softwareentwicklung" (PSE), "Teamprojekt Softwareentwicklung" (TES), "Web-Anwendungen und Serviceorientierte Architekturen (I)", and "Web-Anwendungen und Serviceorientierte Architekturen (II)". In the case study, the participants have to develop an excerpt of the authorization artifacts of CarRentalApp. The excerpt is limited to the use cases *Add Car to Fleet*, *Rent a Car*, *List Customer Rentals*, and *List Car Rentals*.

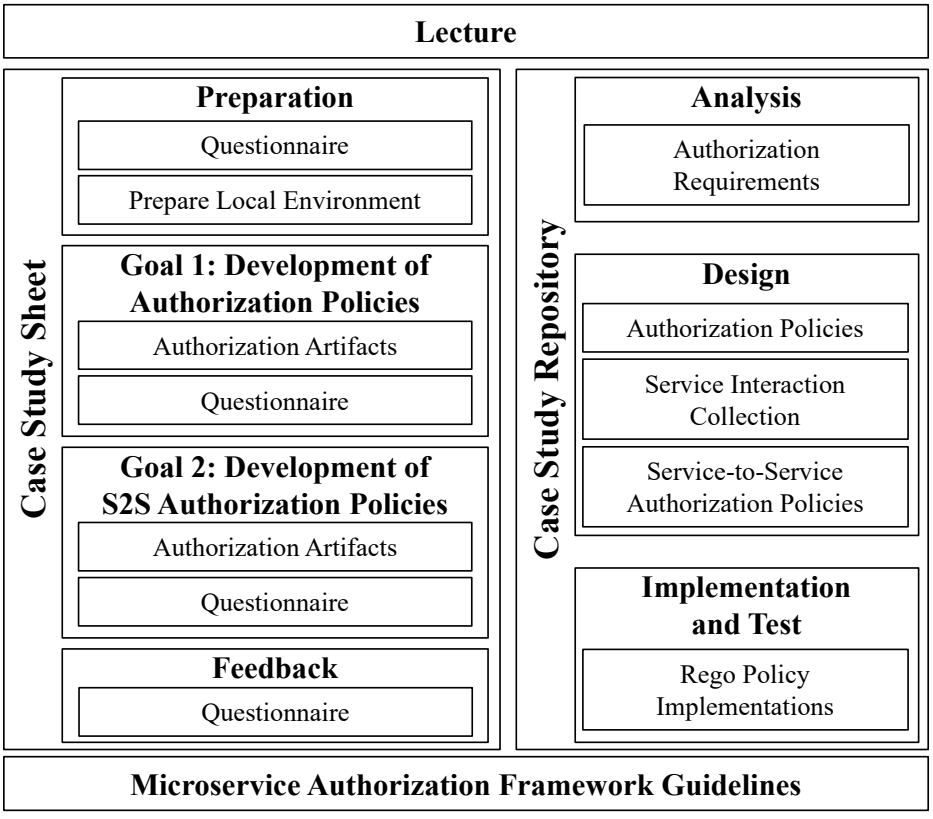


**Figure 8.7:** Setup of the Case Study

Figure 8.7 presents an overview of the case study's setup. First, the participants are asked to

attend a lecture, which provides an overview on authorization and introduces the concepts of MAF. Subsequently, each participant receives a case study sheet (see Appendix A.5.2) and a case study repository and is asked to perform the case study individually at home in a personal environment. The case study sheet structures the case study for the participants into four phases, indicates the task descriptions, and informs the participants about the expected duration and compensation of the case study. In the first phase, the *Preparation* phase, the participants are asked to complete a general questionnaire about their academic background and their overall knowledge of authorization and microservices. In addition, the participants are instructed to prepare their local development environment required for the subsequent phases. Therefore, the participants must download and set up the case study repository, which is described further in the next paragraph. Following the preparation, the participants are prompted to develop the authorization artifacts for the selected use cases in the *Development of Authorization Policies* phase. This includes the authorization requirements, the authorization policies, and the Rego policy implementations. Following the creation of the authorization artifacts, the participants have to fill out a questionnaire related to the artifacts. In the subsequent *Development of S2S Authorization Policies* phase, the participants are required to create the S2S authorization artifacts. Similarly, a questionnaire related to the S2S artifacts is provided to the participants. Finally, in the *Feedback* phase, participants are required to answer a feedback questionnaire to evaluate the overall approach provided by MAF. After the participants finished the case study, the participants are asked to upload the case study sheet containing the answered questionnaire and the case study repository to an online file storage. This upload happens anonymously so that no information can be obtained about the participants and ensure privacy.

Next to the case study sheet, participants are equipped with a complementary case study repository through the GitLab platform at Karlsruher Institut für Technologie (KIT). By this, participants should be supported in the creation of the respective authorization artifacts. The repository is structured into sections along the development phases of analysis, design, and implementation and test. The development artifacts required for the creation of the final authorization artifacts are provided to the participants in the respective sections of the repository. For instance, the analysis section contains the required use cases for the derivation. In addition, examples are provided. For the implementation and test phase, the implementation structure for the Rego policies has already been applied. The participants must only focus on the derivation and implementation of the Rego policies.

The MAF guidelines provided to the participants complement the case study sheet and case study repository. For each authorization artifact, a detailed guideline describing the derivation of the respective artifact is provided. The content of the guidelines is aligned with the content presented in Chapters 5 to 7. In addition, the guidelines entail an example from the CarRentalApp. Thereby, the phases of the case study are linked to the outlined contributions. After finalizing, the results are evaluated by the author.

The participants are compensated with 10 hours that can be credited to the attended lecture. These

10 hours are divided into 5 hours of active participation (i.e., performing the tasks) and 5 hours of retrospect and discussion in their respective thesis. This information is stated in the case study sheet distributed to the participants.

### 8.4.3 Results

This section provides the result of the case study. The case study was performed with 15 participants. The results are structured along the case study sheet. First, an overview of the participants is provided. Subsequently, the results of Goal 1 and Goal 2 are presented. Finally, the general feedback of the participants towards MAF is presented. Unfortunately, due to data collection issues, the questionnaire answers by one participant have been fully lost. One participant only answered the first questionnaire but failed to answer the remaining questionnaires. Therefore, the first questionnaire has $n = 14$ answers while the remaining questionnaire have $n = 13$ answers. However, all authorization artifacts are available ($n = 15$).

#### Overview of Participant Sample

The participants are mostly enrolled as students at KIT. 50 % of the students are currently in their Bachelor's studies in computer science or information science. 35,71 % are Master's students studying computer science. 14.29 % of participants selected the option *other*. Their indicated experience with relevant concepts of the case study is presented in Figure 8.8. The experience is measured on a 5-point Likert scale, ranging from "Very Unfamiliar" (value 1) to "Very Familiar" (value 5). When asked about their familiarity with the development of microservice-based applications, one participant is very familiar and three participants are familiar with microservice-based applications. The majority is somewhat familiar or unfamiliar with microservice-based applications (mean Likert value of 2.79).

A similar picture is drawn when it comes to the familiarity with ABAC and the policy language Rego. The majority of participants are unfamiliar or very unfamiliar with ABAC, while two participants are familiar with ABAC (mean Likert value of 2.43). This can also be seen in the familiarity with the Rego policy language used in the case study. Most participants are unfamiliar with the policy language, with two participants being very familiar with the policy language (mean Likert value of 2.29).

In addition, the participants stated if they have implemented authorization in an application, and if so, how they implemented the authorization. 50 % of participants have never implemented authorization in an application. The remaining 50 % implemented authorization, e.g., with an earlier version of contribution C1 during a previous semester or inside the application logic.

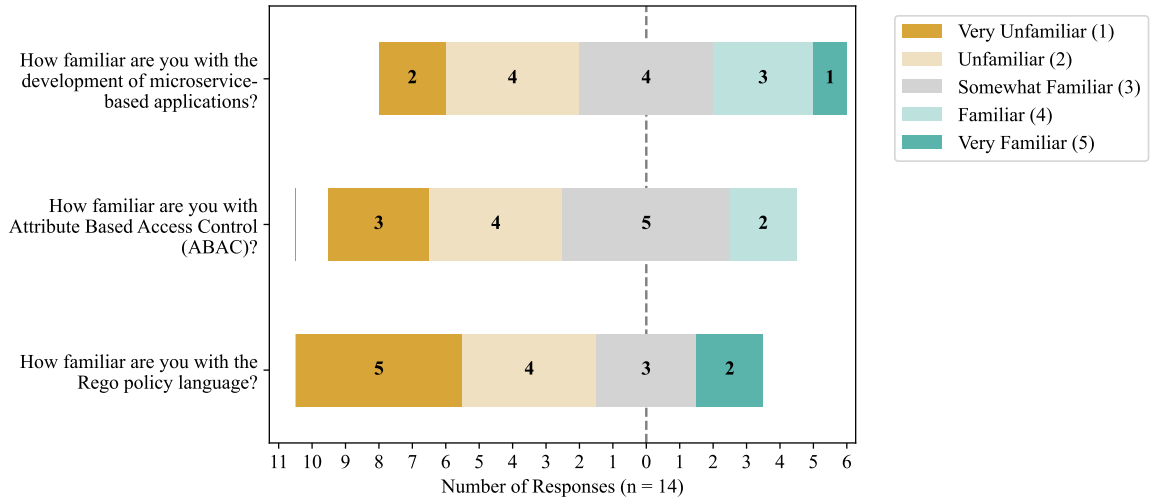


**Figure 8.8:** Overview on Participant Experiences

### Goal 1: Development of Authorization Policies

The first part of the case study is the development of authorization policies based on the selected four use cases. Thus, each participant had to create four authorization requirements, four authorization policies, and implement four Rego policies. The results of Goal 1 are summarized in Figure 8.9 and Figure 8.10. Figure 8.9 presents the correctness of the created authorization artifacts in percent. Figure 8.10 depicts the answers of the subsequent survey on a five-point Likert scale. The results will be discussed by answering the questions Q1.1 to Q1.4.

In total, the 15 participants created 60 authorization requirements. Out of the 60 authorization requirements, a total of 71.67 % percent of authorization requirements were coded as correct. 28.33 % of the authorization requirements contained an error, rendering the authorization requirement incorrect. When investigating the incorrect authorization requirements, the majority of them contains an error regarding the detection of the conditions, i.e., there were too many or not enough conditions. In 56.25 % of these cases, participants included conditions that are not relevant for authorization in an authorization requirement. For instance, the condition CarIsAvailable is not relevant for the authorization logic. In another 37.5 % of the incorrect cases, the participants had the opposite error by missing a condition. For example, missing the condition that a fleet manager should only be able to list the rentals in their fleet. In 6.25 % of the incorrect cases, the participants detected the wrong object.

The majority of the incorrect authorization requirements can likely be linked to the fuzziness in the use case description. This in turn makes the identification of the correct subject, action, object, and conditions inherently complex for an inexperienced developer. While additional conditions might

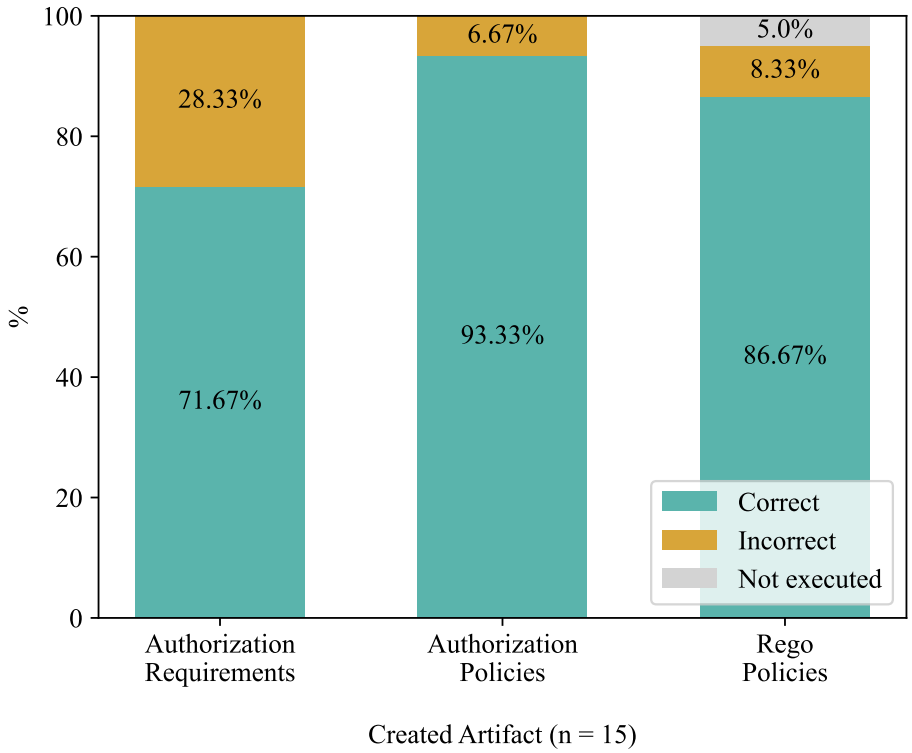


**Figure 8.9:** Correctness of Created Authorization Artifacts

lead to an authorization overhead, missing conditions might result in a lack of authorization, which should be avoided.

Next, for every authorization requirement, an authorization policy must be created by the participants. Compared to the authorization requirements, the number of authorization policies coded as correct increases to 93.33 %. Only 6.67 % of authorization policies are coded as incorrect. Since the authorization artifacts built upon another, authorization requirements that have been specified as incorrect (i.e., due to missing/additional conditions) can still be transformed into correct authorization policies. The authorization policies coded as incorrect contain a false object (i.e., Car instead of CustomerService) or an additional condition not present in the respective authorization requirement (i.e., evaluation of rental time). Since the majority of authorization policies are created correctly, the incorrect authorization policies likely occur due to the participant not closely following the guidelines.

The results of the implementation of the Rego policies are similar to the authorization policies. 86.67 % of the Rego policies have been marked implemented correctly by the author. This includes the Rego policy and the required Rego rules following the structure proposed in Section 5.3. 8.33 % of the policy implementations are incorrect and the result of implementing the wrong object or a

missing implementation of a required condition. The remaining 5.0 % of the policies have not been implemented by the participants.

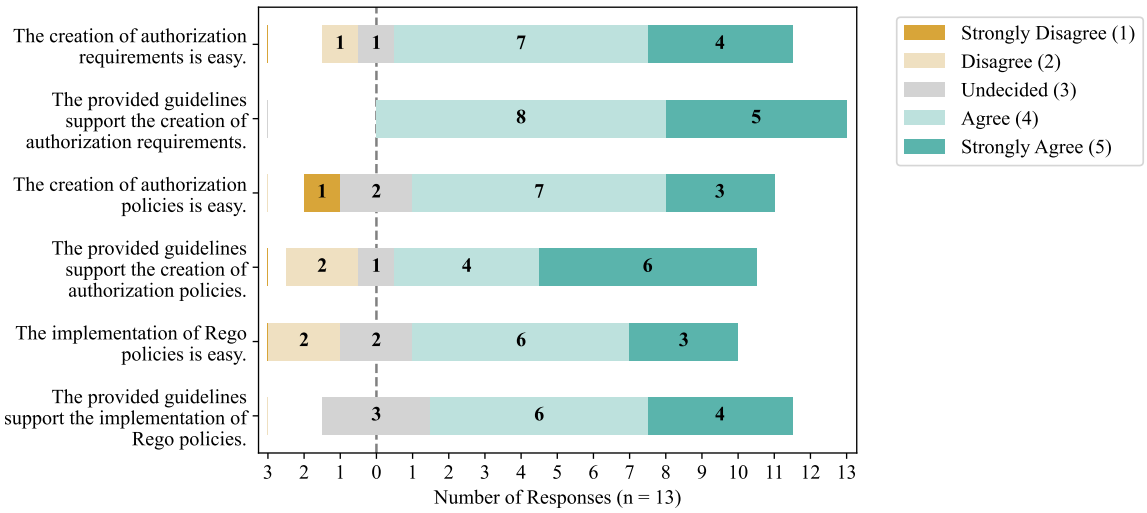


**Figure 8.10:** Survey on Artifact Creation and Guideline Support

Finally, the participants evaluated the support of MAF towards the development of authorization policies. Therefore, the participants had to assess the overall perceived ease of the artifact creation and the perceived support of the respective guidelines. The answer options range from "Strongly Disagree" (value 1) to "Strongly Agree" (value 5). The results are presented in Figure 8.10. The participants assessed the perceived ease of the creation of authorization requirements, authorization policies, and policy implementation with a mean Likert value of 4.08, 3.85, and 3.77 respectively. This indicates an overall high ease of creation of the artifacts by the participants. One participant disagreed with the perceived ease of the creation of authorization requirements. One participant strongly disagreed with the perceived ease of the creation of authorization policies, and two participants disagreed with the perceived ease of the creation of the implementation of Rego policies. The creation of the authorization artifacts is supported by the respective guidelines. Overall, the participants assessed the support provided by guidelines for authorization requirements, authorization policies, and Rego implementations with a mean Likert value of 4.38, 4.08, and 4.08 respectively. This indicates that MAF supports the software engineers with the creation of authorization artifacts.

### Summary Goal 1

The goal G1 targets the applicability of the approach presented in Chapter 5 when provided to software engineers. As the results presented in the previous section suggest, overall, most software engineers (i.e., students) were able to create the authorization artifacts. However, there are challenges when it comes to the creation of the authorization artifacts. The primary challenge is the extraction of the

authorization knowledge from the functional requirements. In this case study, the participants had to extract the knowledge from use cases. This is inherently complex, as the identification if something is relevant for authorization can be subjective and thus dependent on the experience of a developer. To address this uncertainty, the guideline should be revised to focus more on the identification of authorization elements, including more detailed examples.

### Goal 2: Development of S2S Authorization Policies

The second part of the case study focuses on the development of the S2S authorization policies. The participants have to create the authorization artifacts based on the use case *Add Car to Fleet*. This includes the entries in the service interaction collection, the creation of S2S authorization policies, and the S2S policy implementation in Rego. The results of Goal 2 are summarized in Figure 8.11 and Figure 8.12. Figure 8.11 presents the correctness of the created artifacts. Figure 8.12 details the answers of the subsequent survey on a 5-point Likert scale.

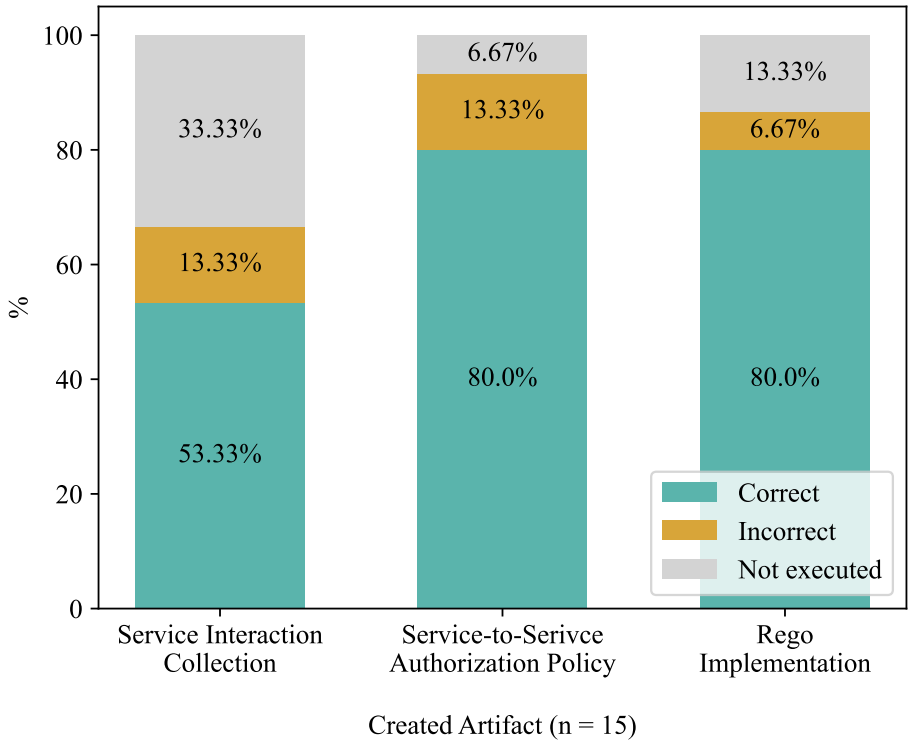


**Figure 8.11:** Correctness of the Created S2S Authorization Artifacts

Figure 8.11 depicts an overview of the correctness of the development of the S2S authorization artifacts. Of the 15 participants, 53.33 % completed the service interaction collection correctly. 33.33 % did not create an entry in the table. This might be due to an ambiguous task description, as the

service interaction collection was only mentioned in the respective guideline but not explicitly stated in the task description. Two participants detected the wrong interactions, added unnecessary interactions, or referenced the wrong authorization requirement. However, 93.33 % of participants performed the derivation of the S2S authorization policy. 80 % derived the correct policy. Two participants (13.33 %) created incorrect S2S authorization policies. Both of them selected the wrong action and one participant (6.67 %) selected the wrong object. In both cases, the action/object from the initial API request was copied. One participant did not create the S2S authorization policy.

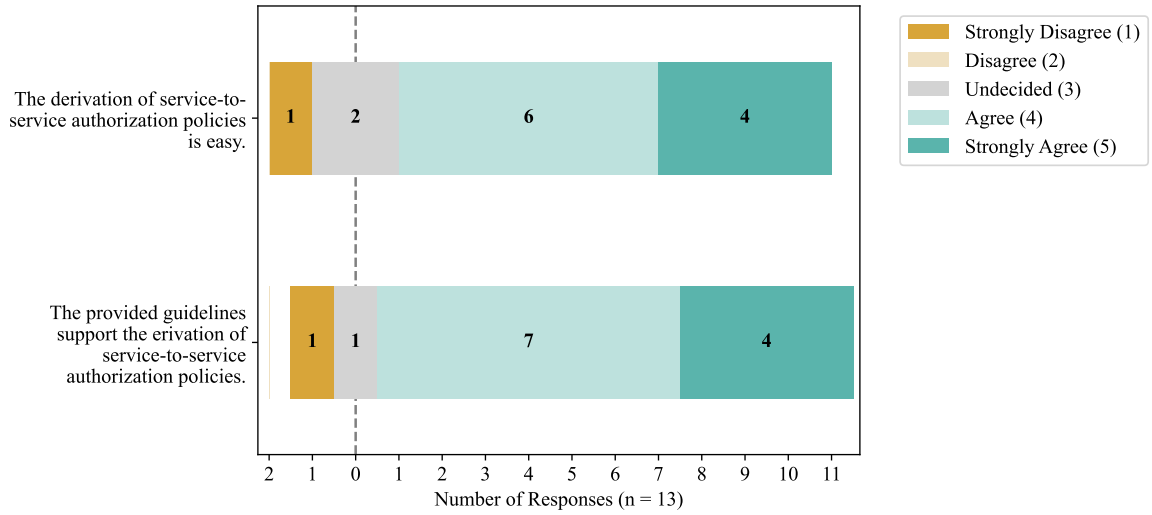


**Figure 8.12:** Survey on S2S Authorization Artifact Creation and Guideline Support

To answer Q2.1, in 53.33 % of cases, the S2S requests have been correctly identified, and the authorization policies have been correctly derived. However, the amount of nonexistent entries in the service interaction collection and obvious mistakes in the derived authorization policies indicate a problem with the provided guideline. The majority of the S2S authorization policies (80.00 %) have been implemented correctly by the participants. One participant (i.e., 6.67 %) implemented the policy correctly, but stored it in the policies' folder of FleetManagement instead of RentalManagement. Finally, two participants did not implement the Rego policy. Using MAF, the S2S authorization policies can largely be implemented correctly, answering Q2.2.

The results for Q2.3 are depicted in Figure 8.12. Again, the answer options range from "Strongly Agree" (value 1) to "Strongly Disagree" (value 5). Overall, the participants evaluated the perceived ease of the derivation of the S2S authorization policies, with a mean Likert value of 3.92. This is worse than the creation of the authorization artifacts described in Goal 1. A similar result is presented by the perceived support provided by the guideline, with a mean Likert value of 4.00. In both cases, one participant strongly disagreed with the respective questions. The other participants assessed the statements largely by (strongly) agreeing or being undecided. Nonetheless, the guideline should be improved, e.g., to clearly state the final location of the implemented policy.

### Summary Goal 2

Goal 2 evaluates the applicability of the derivation and implementation of S2S authorization presented in Chapter 6. The identification in the service interaction collection has not been performed by all participants. This indicates a problem with the guideline or an unclear task description. The artifacts created were predominantly marked as correct. The errors that led to incorrect artifacts could have been caused by a lack of experience or carelessness by the participants. Overall, the goal is considered to be fulfilled but suggests improvements that are required to be performed to the guidelines.

### Feedback Questionnaire

Finally, the participants received a feedback questionnaire in which they had to assess features and characteristics of the overall approach. The results are depicted in Figure 8.13. The questionnaire used a 7-point Likert scale, in which 1 refers to the negative answer and 7 to the positive answer. The questionnaire considers understandability, complexity, clarity, simplicity, completeness, support, and traceability of artifacts.

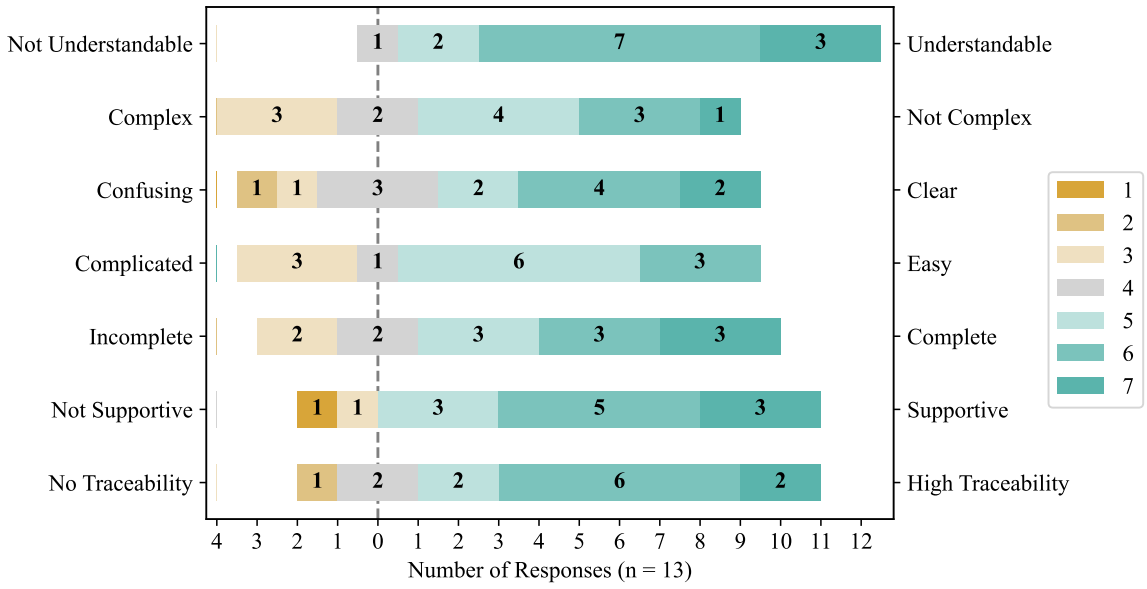


**Figure 8.13:** Overall Feedback on MAF

The participants determined the concepts of MAF as understandable, with a mean score of 5.92. With a mean value of 4.77, the complexity tends to be described as not complex. However, three participants find MAF rather complex, two rather confusing, and three rather complicated. Contrary, most participants determine MAF as rather clear (mean 5.0) and easy (mean 4.69). Overall, the participants find MAF as rather complete (mean 5.23) and supportive (mean 5.38). Again, two

participants determined MAF as not supportive. Finally, the participants observed the traceability of the authorization artifacts, with a mean value of 5.38.

### 8.4.4 Threats to Validity

We discuss the threats to the validity of the applicability of MAF in this section. Again, to structure the threats to validity, we use the threats which can be applied to case studies as proposed by Wohlin et al. [WR+12].

**Construct Validity** We assessed the metrics in the form of both, objective and subjective metrics. Therefore, we focused on correctness as a necessary condition to ensure that the presented processes lead to the anticipated authorization. However, other metrics such as the time span required to create an artifact might be of interest. A second threat to the construct validity could stem from the evaluation apprehension and social desirability bias. For instance, participants might have been (unconsciously) biased towards providing a more favorable assessment of the overall approach due to the attendance of the respective lecture. To mitigate this bias, the participants had to submit their results anonymously.

**Internal Validity** The primary threat to internal validity is that of uncontrolled factors influencing the outcomes when using MAF. This is for example the case as the (limited) previous knowledge of students could be an important factor influencing the evaluation of MAF. In the case study, the knowledge about microservice-based applications and ABAC was limited. Whether and how the level of experience affects the correctness and the perceived ease cannot be determined with the current design of the case study. For instance, only two participants were familiar with ABAC. Another threat to validity is the duration of the case study, which was estimated to 5 hours. This can have effects on the overall results, e.g., if the participants lose focus, split up tasks, or carry out the case study at several points of time. In addition, it is unclear if learning effects between the application of the guidelines for different use cases exist. To mitigate this risk, we did not explicitly specify the order in which the authorization artifacts for the use cases had to be solved by the participants. A last threat is the design of the instrumentation. Specifically, the perceived clarity of the presentation of the guidelines, including the examples, might affect the outcome.

**External Validity** The external validity is threatened by the selection of participants. The participants are composed of students without well-founded experiences in software development or security. While Tichy argues that (computer science) students should be considered as participants due to their training, there might be differences to professionals [Ti00]. For instance, a developer with a extensive security knowledge may detect the required authorization conditions and be able to

differentiate conditions considered as business logic. Besides, the use cases of the CarRentalApp might not fully represent the characteristics of an application in the industrial context. In an external (i.e., industrial) context, the application may be more complex, which can result in incompatibilities regarding the introduced guidelines (e.g., use of different API paradigms). However, the CarRentalApp uses state-of-the-art technologies, which are also employed in industry.

**Reliability Validity** The use of a single author to code the results poses a threat to reliability validity. Specifically, the metric of correctness, i.e., the percentage of artifacts that have been created correctly by a participant, is assessed in the case study. Despite a pre-defined set of criteria to code the correctness, as introduced above, other researchers might evaluate the correctness differently. This can lead to a different outcome. For example, a typo in the name of an object was coded as correct since it does not influence the semantic understanding of the participant. Other researchers might code this as incorrect.

### 8.4.5 Summary of Type 2 Validation

The goal of the Type 2 validation is the evaluation of the applicability of the MAF. Therefore, a case study has been performed with 15 participants in which the participants had to implement authorization artifacts for selected use cases of CarRentalApp. The results of the case study are interpreted using the GQM approach. To evaluate if an artifact has been created correctly by a participant, the implementation of CarRentalApp created during the Type 1 validation is used as a reference implementation. Overall, the participants have mostly created authorization artifacts correctly by using the guidelines of MAF. This indicates the applicability of MAF. By addressing goal G1, the systematic development of authorization policies is demonstrated. The second goal G2, demonstrates the systematic development of S2S authorization policies by participants. However, there are limitations in the creation of authorization requirements or the identification of S2S requests. The guidelines should be revised accordingly (e.g., by introducing additional examples) in future work.

## 8.5 Summary

In this chapter, the contributions of MAF have been validated. Three forms of validation have been applied to MAF. First, the Type 0 validation compares the overall feasibility of MAF to a requirements catalog based on related work created in Section 3.1. Second, the Type 1 validation verifies the suitability of MAF, by applying MAF to two case studies, CarRentalApp and an excerpt

of TrainTicket. Finally, Type 2 assesses the applicability of MAF by performing a case study with 15 participants.

Despite a rigorous application of the methods and chosen sample of computer science students, the validation is subject to limitations. Specifically, the external validity and generalizability of the findings are limited due to the lack of implementation in a field environment. Future work should perform a field study in an industrial context with experienced developers. The systematic integration of authorization provided by MAF should be compared to an existing approach implementing authorization in a microservice-based application. This will provide more detailed insights into the versatility of MAF in environments that differ from the one presented in this thesis.

# 9 Conclusion and Future Work

Authentication and authorization are essential to secure an application [SS94]. With protocols such as OpenID Connect (OIDC) or Security Assertion Markup Language (SAML), well-established solutions for authentication exist. On the authorization side, OAuth2.0 is commonly used to perform coarse-grained authorization, e.g., to authorize whether a service can be accessed or not. However, the aspect of authorization on a fine-granular level is still an active research area in microservices [AC22]. This thesis contributes to the area by providing the Microservice Authorization Framework (MAF), which supports developers with the systematic integration of fine-grained authorization using authorization policies. This chapter provides a conclusion to the development of MAF in Section 9.1. Section 9.2 introduces future work in this field.

## 9.1 Conclusion

The core contribution of this thesis is the MAF. The goal of creating MAF is to support developers with the integration of fine-grained authorization using policies into their microservice-based applications. This requires a holistic consideration of authorization throughout all development phases of a Software Development Life Cycle (SDLC). Related work presented in Chapter 3 primarily focused on certain aspects of the development, such as the generation of informal policies based on natural language processing or the implementation of authorization in a microservice-based application. With the help of Attribute Based Access Control (ABAC), this thesis provides authorization for microservices at a fine-granular level. This depends on the availability of the required attributes. Additionally, ABAC allows externalizing authorization logic fully from a microservice. Thereby, a microservice only has to provide its core business logic through an Application Programming Interface (API). This adheres to the principles of low coupling and high cohesion, which allow the microservices to be reused in different scenarios, independent of authorization aspects. Furthermore, the respective technologies used for authorization can be exchanged without having to modify the microservice implementation.

The holistic consideration allows understanding what must be authorized in a microservice-based application and how the authorization can be systematically realized. The following sections address the research questions established in Section 1.3.

### RQ1 - How to systematically integrate authorization into the development of microservice-based applications?

The first research question demonstrates the overarching question of this thesis. By introducing the MAF in Chapter 5 we address RQ1. The MAF provides two viewpoints on authorization in microservice-based applications: First, the derivation and implementation of authorization policies, which is introduced in contributions C1 and C2. Second, the integration of the components required to perform policy-driven authorization into a microservice-based application, presented in contribution C3. The integration is considered throughout all phases of the SDLC. To support the systematic integration, the introduced artifacts build upon another and are integrated into an existing development approach for microservice-based applications. The case study performed with students indicates that MAF provides supportive and understandable guidelines for developers. Further work is required to improve the guidelines and demonstrate the applicability in field contexts.

### RQ2 - How to derive authorization requirements from existing analysis artifacts?

The derivation of authorization requirements shown in contribution C1 introduces an approach to answer RQ2. The contribution includes a process to identify relevant terms and conditions for ABAC and to transfer these terms and conditions into a structured authorization requirement. Templates are used to structure these authorization requirements. According to Hu et al. [HF+14], the authorization requirements can be defined as natural language policies. We propose the use of functional requirements to derive authorization requirements. The functional requirements should define what a user is allowed to do with the system. In this thesis, use cases describe the functional requirements because they allow to include conditions which provide further granularity. In the ideal case, for every functional requirement, an authorization requirement is created. This creates a close relation between the artifacts which are used as the foundation for the subsequent development phases. The derivation of authorization requirements performed with students demonstrated that the derivation from use cases is suitable. However, there are inherent challenges when it comes to the identification of authorization terms in natural language. For instance, the experience of the developer. Future work should investigate the use of a Large Language Model (LLM) to support developers with the identification (see Section 9.2).

### RQ3 - How to implement fine-grained authorization policies?

To address the implementation of fine-grained authorization policies, a systematic process is presented in Chapter 5. The process uses the previously created authorization requirements and transforms them into authorization policies. In the design phase, for every authorization requirement, an authorization

policy is created. In the context of MAF, an authorization policy created in the design phase is independent of a specific policy language. This allows the realization of an authorization policy in different policy languages in the implementation and test phase. The derivation of an authorization policy requires an authorization requirement and the API specification of the respective microservice that is responsible for realizing the functional requirement. By using existing development artifacts such as the API specification, the authorization policies can be created on a fine-granular level. The authorization policies are subsequently implemented in the implementation and test phase. In this thesis, the Rego policy language has been used to implement policies as code. In addition, the policies can be structured into different folders, which reduces code duplications. The applicability of the proposed concepts has been presented in a case study on the CarRentalApp. Employing the structures introduced by the authorization policies in the design as well as the implementation and test phase allows developers to implement the policies without requiring extensive previous knowledge, e.g., of the Rego policy language.

### RQ4 - How to implement service-to-service authorization in a microservice-based application?

The implementation of Service-to-Service (S2S) authorization in a microservice-based application is addressed by contributions C2 and C3. C2 complements C1 by developing S2S authorization policies. Since the decision to use the microservice architecture is made throughout the design phase, the process of developing S2S authorization policies begins in the design phase. The approach considers S2S requests that are the result of the interaction of a user with the microservice-based application. For these S2S requests, the authorization requirements defined in contribution C1 should also hold. This assumption allows the creation of fine-grained S2S authorization policies. In the development of S2S authorization policies, the S2S requests must first be identified. This can be done by analyzing the software architecture and the orchestration definition of the microservice-based application. The S2S requests are documented in a service interaction collection and are associated with an authorization requirement. For every entry in the service interaction collection, a S2S authorization policy is created. A S2S authorization policy is similar to an authorization policy created in C1 but contains rules to identify the involved source microservice. The S2S authorization policies are implemented in the implementation and test phase. Similar to contribution C1, the Rego policy language is used to systematically implement the S2S policy. To support S2S authorization, the microservices of a microservice-based application must propagate the identity of the subject initiating the S2S requests. This allows to forward the user's identity through the S2S requests, allowing fine-grained authorization. However, to support identity propagation, the microservice must be adapted to support identity propagation. In the context of CarRentalApp these changes were minimal. Since the TrainTicket application already supported identity propagation, no additional changes were required. The use of the contributions C2 in the CarRentalApp and TrainTicketApp

demonstrated the applicability of the S2S concepts. However, there are still options for improvements to the respective guidelines to improve the developer experience.

### RQ5 - How to externalize authorization in a microservice-based application?

The externalization of authorization in a microservice-based application is addressed in the integration of authorization in the microservice-based application introduced in C3 in Chapter 7. The authorization integration also follows the phases of SDLC. In the analysis phase, the requirements for the integration of authorization components are elicited. This includes the externalization and the technology selections. In the design phase, the architecture of the microservice-based application is adapted. The concepts of the eXtensible Access Control Markup Language (XACML) reference architecture are applied to the microservice-based application. This includes the placement of the components Policy Enforcement Point (PEP), Policy Decision Point (PDP), and Policy Information Point (PIP), which allows externalizing the authorization fully in a microservice-based application. Furthermore, the load tests conducted in Chapter 8, indicate that the latency introduced by the externalization in the examples of the CarRentalApp and TrainTicket application has increased. However, since the response latency is lower than 100 ms, the application will likely feel fluid for users as outlined by Dean and Barroso [DB13].

### RQ6 - How to decentralize authorization in a microservice-based application?

Similar to the externalization of authorization, the decentralization is addressed in contribution C3 in Chapter 7. In the analysis phase, an integration requirement for the decentralization is created. The design phase considers this requirement by introducing a PEP and PDP per microservice. This enables the deployment of the components together with a microservice in a Kubernetes-based deployment environment as a single, scalable unit. To support the decentralization of authorization, a mechanism to distribute the authorization policies is presented using a Continuous Integration / Continuous Deployment (CI/CD) pipeline. With this mechanism, the authorization policies can be automatically updated at the PDP. Furthermore, a mechanism to collect authorization decisions for auditing purposes is demonstrated. Again, the latency of the decentralized PDPs is presented in load tests for a S2S request in Chapter 8. In this scenario, every microservice has its dedicated PDP. The latency introduced by the authorization components will add up depending on the length of the S2S request chain. While this is inherent to a long S2S request chain (and a reason why this should be avoided [Ne15]), the introduced latency must be considered when developing applications that are time-critical (e.g., due to user interaction).

## 9.2 Future Work

Authorization is an inherently complex subject. MAF primarily considers the development of authorization policies and the integration into a microservice-based application. To further tackle the complexity of authorization, the following paragraphs introduce future research directions for all aspects of this thesis.

### Attribute Management

The foundation for fine-grained authorization using ABAC is the availability of attributes. Without attributes, the conditions of an authorization policy cannot be evaluated. ABAC accesses the attributes through a logical component introduced as PIP. In this thesis, the PIP accesses the backing services of a microservice directly. This can be a feasible solution in a (relatively) small environment. However, in a bigger environment, managing and accessing the attributes becomes more complex. This also includes the topicality of attributes and privacy concerns. For instance, if an attribute is changed by an European employee, how long will it take for the attribute change to have an effect on employees in Australia (e.g., replication time)? What can happen in the meantime? Future work should investigate the management and distribution of attributes. In some cases, caching attributes might also be a viable option to improve overall performance by removing communication. Initial technical approaches in this direction are already being made by the Open Policy Agent Administration Layer (OPAL) [Pe-OP].

### Alternative ABAC Models

Unfortunately, ABAC does not have a standardized model (see Section 2.5). The MAF presented in the thesis relies on the ABAC model for web services Yuan and Tong [YT05]. As highlighted by Servos and Osborn, various other (domain-specific) ABAC models exist [SO17]. Future research should investigate how different ABAC models can be used in the context of web applications. In addition, research should explore how these ABAC models impact the concepts introduced by MAF. Furthermore, the aspect of delegation in the context of ABAC should be researched.

### Formalization of Artifacts

The artifacts created by MAF do not have a formal specification. Instead, the artifacts such as authorization requirements or authorization policies are pragmatically specified following a template. While this provides flexibility when creating the artifacts, determining if an artifact is correct or false becomes difficult. To support the uniform development of authorization artifacts in a larger

organization, the artifacts should be formalized in future work. A simple example for the formalization using an Augmented Backus Naur Form (ABNF) is presented in Appendix A.1.

### Automated Creation of Authorization Artifacts

The related work presented in Chapter 3 focuses on the automated creation of authorization artifacts. The authors argue that the creation of authorization policies is a labor-intensive task [LC+21]. These approaches include natural language processing of analysis artifacts or the derivation of authorization policies based on source code. However, as elaborated in Chapter 3, the approaches are limited by the granularity they provide.

With the introduction of LLMs such as GPT or LLAMA [ZZ+24], the possibilities in the processing of natural language in a software engineering context have dramatically increased. LLMs are already used to support developers with the generation of source code (e.g., using GitHub Copilot [YO+23]) or the extraction of knowledge from user stories (e.g., class diagrams [LK+24]). Future work should investigate how these models can be used to derive authorization artifacts. This concept has also been proposed by Martinelli et al. [MM+24].

### Testing of Authorization Policies

Chapter 5 introduces the systematic structure for the implementation of authorization policies using the Rego policy language. Section 5.3.4 demonstrates a simple unit test for a Rego policy. Future work should further investigate the development of a test concept, which allows to systematically create tests for authorization policies. This includes the derivation of test data, systematic creation of unit tests, and creation of integration tests including the authorization components. The testing of authorization policies should also be an essential part in the release process for authorization policies.

### Performance Impact and Resource Consumption

As presented in Section 8.3, introducing the authorization components into a microservice-based application has an impact on latency. Future work should investigate the impact on performance, not just regarding latency. This includes a monitoring of the resources required by the additional authorization components. The resources assigned to the authorization components must be coordinated with the load a microservice can handle to reduce the overall amount of required resources. In this context, a cost-benefit analysis should be performed, comparing the benefit created by the

policy-driven authorization introduced by MAF with the additional costs created in an industrial context.

# A Additions

## A.1 Formalization of Authorization Artifacts

### Authorization Requirements

The ABNF allows defining a language using a set of rules and elements [CO08] and has been used to structure requirements in other contexts [GZ+21; MC+22]. Listing A.1 presents an exemplary ABNF of a possible definition for authorization requirements. The presented ABNF defines five rules which are non-terminal symbols depicted in bold formatting. The primary rule called *Authorization-Requirement* depicted in line 1 is the start symbol of the ABNF. This rule defines the structure of the authorization requirement. The rule concatenates strings (i.e., terminal symbols such as *Subject*, *can perform action*) with the rules subject, action, and object. If there are any conditions, the notation *0*1("If" condition)* in line 2 allows defining conditions using the respective rule.

```
1 AuthorizationRequirement = "Subject" subject "can perform action" action
2                            "on object" object 0*1("IF" condition)
3 subject     = 1*VCHAR
4 action      = 1*VCHAR
5 object      = 1*VCHAR
6 condition   = 1*VCHAR
7 condition   =/ condition "AND" condition
```

**Listing A.1:** Augmented Backus Naur Form for Authorization Requirements

The rules subject, action, object, and conditions (lines 3 to 6) each use the rule *VCHAR*, which allows defining an arbitrary long set of characters. The rule *VCHAR* is a core rule specified by the ABNF and allows writing all visible printing characters [CO08]. By using the notation *1*VCHAR*, each rule must result in at least one character. Thus, the rules are not allowed to create empty strings (i.e., string with length of 0). The use of *VCHAR* allows, for example, to define actions such as view, delete, or modify or arbitrary objects (e.g., document, project). In addition, a condition arbitrary complex (e.g., *object is blue*). Furthermore, the ABNF allows defining alternative rules, which are depicted by *=/*. The rules in line 7 allow writing more complex condition statements by introducing a logical *AND* statement. An *AuthorizationRequirement* can have none, one, or multiple condition statements.

### Authorization Policies

Listing A.2 presents an exemplary ABNF for an authorization policy. The result of the ABNF is a sentence containing detailed information from design artifacts (e.g., class diagram, API specification). Compared to the authorization requirement presented in Listing A.1, the ABNF for an authorization policy contains additional rules (i.e., non-terminal symbols) which are highlighted in bold formatting. The rule *AuthorizationPolicy* (lines 1 and 2) is the start symbol and requires the rules *action*, *object*, and *condition*.

```
1  AuthorizationPolicy = "subject can perform action"  action
2                        "on" object "IF" condition
3  action = "GET" / "POST" / "PUT" / "PATCH" / "DELETE"
4  object =0*1("every object in") 1*VCHAR
5
6  condition = ("subject" / "object" / "environment")"." attribute operator
7            ((("subject" / "object" / "environment")"." attribute) / value)
8  condition =/ condition "AND" condition
9  attribute = 1*VCHAR
10 operator = "<" / "<=" / "==" / ">" / ">=" / "is" / "not" / "contains" / ..
11 value = 1*VCHAR
```

**Listing A.2:** Augmented Backus Naur Form for Authorization Policies

Compared to the authorization requirement, the rule *action* (line 3) is no longer a **VCHAR*. Instead, the rule *action* contains the HTTP operations as the terminal symbols. The rule *object* (line 4) is consistent with the authorization requirement and allows a definition of an object based on a arbitrary amount of characters of a length larger than 1 (i.e., *1*VCHAR*). Similar to the action, the object must be accessed through a HTTP path which is defined in the respective API specification. The definition of the object further depends on whether a single object or a set of objects is accessed. This is denoted in the rule *object* presented in line 4. RESTful APIs allow to either access a set of objects or a single object. A set of objects is denoted through the plural of the object name (e.g., /rentals) and a single object is accessed through an identifier (e.g., /rentals/{id}). If a request is performed on a set of objects, *every object in object* is used.

The rule for a condition is presented in line 6. A *condition* is structured to compare and attribute belonging to a subject, an object, or the environment, to either another attribute or a certain value. Attributes and values (lines 9 and 11) are again characters with a length larger than 1. To perform a comparison, the rule *operator* is introduced in line 10 which contains a set of logical operators. Similar to the authorization requirements, multiple conditions can be concatenated using a logical *AND* (line 8).

## A.2 Envoy Input

```
{
  ...
  },
  "parsed_body": {
    "location": "Karlsruhe",
    "vin": { "vin": "JH4DB1561NS000565" },
    "fleetId": "id-a"
  },
  "parsed_path": ["fleetmanagement.FleetService", "AddCarToFleet"],
  "parsed_query": {},
  "truncated_body": false,
  "version": { "encoding": "protojson", "ext_authz": "v3" }
}
```

**Listing A.3:** Example Envoy Input for Use Case "Add Car to Fleet"

## A.3 Implementation in Further Policy Language

### Casbin

This section provides a brief overview of the implementation of authorization policies in Casbin [CO-Doc]. Casbin is an open-source authorization framework, which can also be employed to implement authorization policies created during the design phase in Chapter 5. Listing A.4 shows the Casbin policy model for the micorservice FleetManagement. The model defines the structure and behavior of a policy. The request definition (lines 1 and 2) defines how the incoming request is structured. The structure of the policy is specified in the policy definition (lines 3 and 4). In this case, the policy is structured into role, object, action, and condition. The policy effect (lines 5 and 6) specify how many policies must be evaluated to true to allow the request. Finally, the matchers (lines 7 and 8) define the policy template. In this case, the matcher is a boolean operation comparing the request with the policy.

```
1 [request_definition]
2 r = sub, input
3 [policy_definition]
4 p = role, obj, act, cond
5 [policy_effect]
6 e = some(where (p.eft == allow))
7 [matchers]
8 m = r.input.parsed_path == (p.obj, p.act) && p.role in r.sub.roles && eval
      (p.cond)
```

**Listing A.4:** Casbin Policy Model for FleetManagement

Table A.1 depicts the Casbin policies which follow the structure defined in line 4 in Listing A.4. The content of these Casbin policies can be adopted from the authorization policies created in the design phase. However, Casbin does not provide the flexibility to retrieve attributes similar to Rego. Therefore, Casbin must be extended to provide functions that can be accessed by the Casbin policy. For example, Listing A.5 shows an excerpt for the condition *fleetManagerAssignedToFleet* used in Table A.1. The conditions can be implemented in a Golang function (lines 1 to 6) and registered to the Casbin engine (lines 8 to 11).

| Policy Definition | Subject | Action | Object | Condition |
|---|---|---|---|---|
| p | fleet-manager | fleetmanagement.-FleetService | AddCarToFleet | fleetManager-Assigned-ToFleet(r.input.-parsed_body.fleetId, r.sub.sub) |
| p | fleet-manager | fleetmanagement.-FleetService | ListCarsInFleet | fleetManagerAssigned-ToFleet(r.input.-parsed_body.fleetId, r.sub.sub) |
| p | fleet-manager | fleetmanagement.-CarService | ViewCarInformation | carInFleetOf-FleetManager(r.input.-parsed_body.vin.vin, r.sub.sub) |
| p | fleet-manager | fleetmanagement.-FleetService | RemoveCar-FromFleet | carInFleetOf-FleetManager(r.input.-parsed_body.vin.vin, r.sub.sub) |

**Table A.1:** Casbin Policy Definition for FleetManagement

```
1 FleetManagerAssignedToFleetFuncWithDB(db *dbfleets.DatabaseConnection)
      func(args ...interface{}) (interface{}, error) {
2   return func(args ...interface{}) (interface{}, error) {
3     fleetID := args[0].(string)
4     sub := args[1].(string)
5     return fleetManagerAssignedToFleet(db, fleetID, sub), nil
6   }
7 }
8 ------
9 func  (a *AuthorizationServer) registerDataFunctions(e *casbin.Enforcer) {
10    e.AddFunction("fleetManagerAssignedToFleet", data.
          FleetManagerAssignedToFleetFuncWithDB(&a.fleetdb))
11    e.AddFunction("carInFleetOfFleetManager", data.
          CarInFleetOfFleetManagerFuncWithDB(&a.fleetdb))
12 }
```

**Listing A.5:** Condition Implementation for Casbin Policy

## A.4 Verification of Tokens

As introduced in Section 7.3, the template contains the pre-defined mechanism to validate and decode access tokens. Access tokens are used to encode the subject attributes. Tokens can be structured in various formats, such as a JSON Web Token (JWT) which is the result of the use of an authentication using OIDC [SB+14] or a SAML assertion which results from an authentication using SAML [HM05]. The implementation template includes the mechanism to decode and verify a JWT.

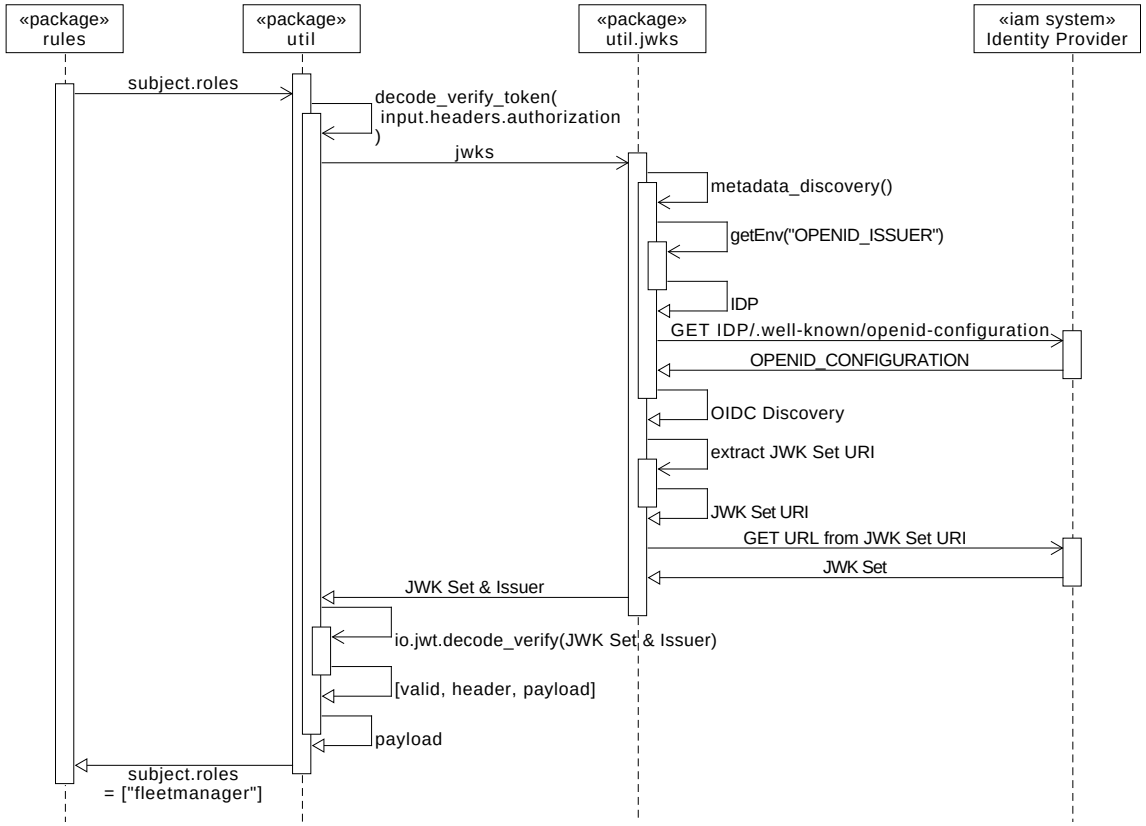


**Figure A.1:** Sequence for Token Verification

JWTs are structured into a header, a payload, and a signature. The token is base64 encoded. The header describes how the JWT is encoded. The payload includes the content of the token, i.e., the subject attributes. The signature is encoded using a private key using the algorithm described in the header. Figure A.1 depicts the process to verify and decode the access token using JWT format. The util package provides the variable subject, which can be used by the Rego rules located in the package rules. The process is initiated if the subject variable is accessed, e.g., by accessing the roles through *subject.roles*.

To verify the access token, the public keys from the issuer must be retrieved by OPA. Using the keys, the signature of the token can be cryptographically verified. To access the public keys, the metadata

from the OIDC provider must be retrieved. The metadata document describes the configuration of an OpenID provider. Among others, the metadata document contains a JSON Web Key (JWK) which contains the required keys. This process is performed by the function *metadata_discovery()* in the util.jwks package. From the discovery document, the URL to the JWK set is extracted and subsequently retrieved. The JWK set is then used by the built-in Rego function *decode_verify* to verify if the signature is valid and to decode the payload of the token. The content of the access token is then available in the variable subject. When accessing the variable *subject*, this process is conducted once.

Since the metadata document of the OpenID provider and the JWK set will not change frequently, the results can be cached. This reduces the amount of requests which must be performed for every request, reducing the latency of the policy evaluation.

## A.5 Validation

### A.5.1 TrainTicket

```
Title: Pay a Reservation Using an External Payment Provider

Primary Actors: User
Secondary Actors: None

Preconditions:
    - User is logged-in
    - User booked the reseveration to be payed
    - User has not enough money deposited
Postconditions:
    - The reservation is booked, allowing the user to collect the ticket

Flow:
1. Actor searches for their reservation and pays it.
2. System prompts a confirmation dialog.
3. Actor confirms.
4. System shows success pop-up.

Alternative flows:
1a. Reservation already payed.
    1. Actor cannot find any reserverations to pay.
3a. Actor denies.
    1. System closes confirmation dialog and cancels payment.

Information Requirements: None
```

**Listing A.6:** Selected TranTicket Use Case "Pay a Reservation Using an External Payment Provider"

```
---UserPayAReservationUsinganExternalPaymentProvider---
subject user is allowed to perform
action pay on
object reservation if
condition UserHasBookedReservation
```

**Listing A.7:** Authorization Requirement "Pay a Reservation Using an External Payment Provider"

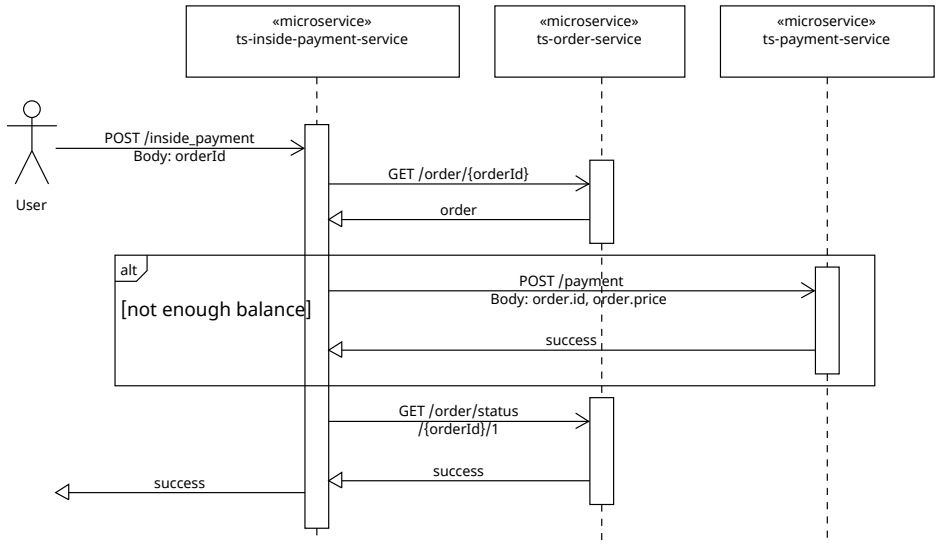


**Figure A.2:** Orchestration Definition "Pay a Reservation Using an External Payment Provider"

```
---OrderService-UserPayAReservationUsinganExternalPaymentProvider---
service can perform
action GET on
object /api/v1/orderservice/order/{orderId} if
source is ts-inside-payment-service and
ROLE_USER in subject.roles and
order(input.orderId).accountId == subject.id
---OrderSerivce-UserPayAReservationUsinganExternalPaymentProvider---
service can perform
action GET on
object /api/v1/orderservice/order/status/{orderId}/{status}  if
source is ts-inside-payment-service and
ROLE_USER in subject.roles and
order(input.orderId).accountId == subject.id
---PaymentService-UserPayAReservationUsinganExternalPaymentProvider---
service can perform
action POST on
object /api/v1/paymentservice/payment if
source is ts-inside-payment-service and
ROLE_USER in subject.roles and
input.userId == subject.id and
order(input.orderId).accountId == subject.id and
```

**Listing A.8:** TranTicket Service-to-Service Authorization Policies

### A.5.2 Goal Quest Metric Plan

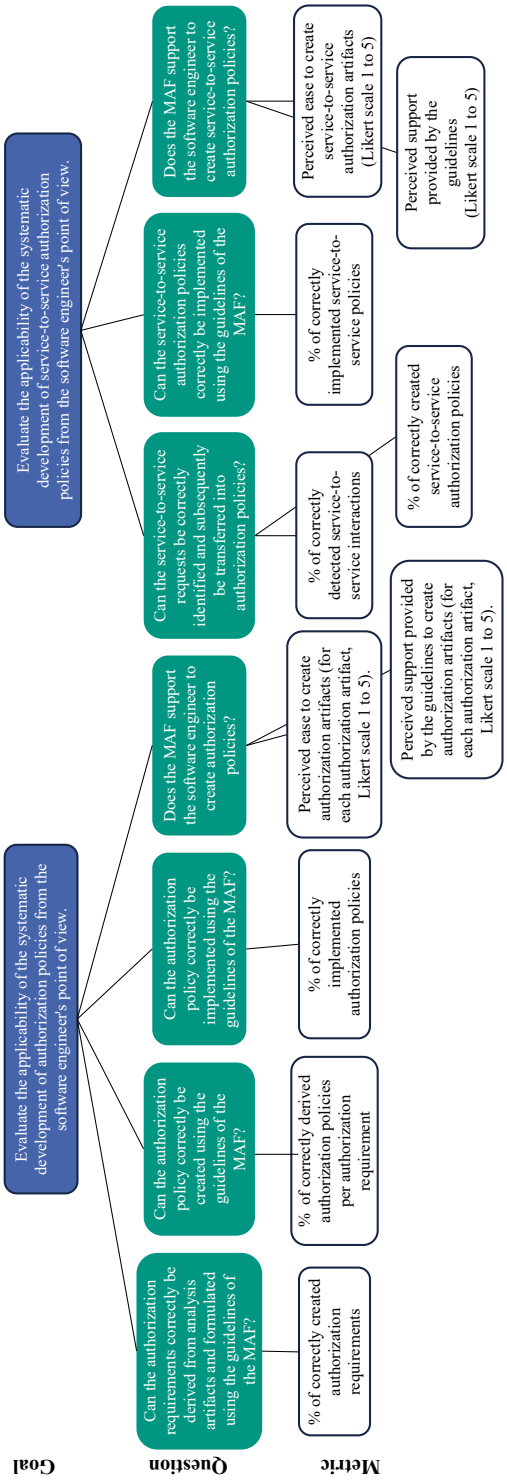


**Figure A.3:** Goal Question Metric Plan Used by Type 2 Validation

### A.5.3 Case Study Sheet


Case Study on Policy-Driven Authorization
in Microservice-Based Applications

Niklas Sänger

## Foreword

Dear participant,

in the context of my research towards a PhD at Karlsruhe Institute of Technology (Faculty for Informatics, Research Group Cooperation & Management, Prof. Abeck) I work on the systematic development of authorization for microservice-based applications. The core contribution of this work is the development of authorization policies, which is evaluated in this experiment.

The data will be collected and evaluated anonymously. The case study consists of three stages. First, the case study is setup by answering a general questionnaire and by preparing a local development environment. Subsequently, the authorization policies are developed. An excerpt of the BestRentalAppV2.0 has been selected to provide use cases. Finally, the case study artifacts are collected and the individual parts of the approach evaluated in a final questionnaire.

Overall, the case study should take no more than 5 hours. After the experiment, the participation of the case study should be described in your practical thesis. Therefore, a section "Project Contribution CaseStudyParticipation" should be created. The participation in the case study can be recorded as 10 hours (5 hours case study participation, 5 hours follow-up in thesis) in the time sheet.

The participation in the case study will have no effect on the final grade of the practical course. The data will be evaluated anonymously. The description of the case study in the practical thesis will be graded like any other project work.

Thank you for taking part in the case study and for completing the evaluation form conscientiously!

Niklas Sänger

## 1 Preparation

1. **General Questions**
   Fill out the survey "Survey Part 1: General Questions" in Section 4.1. Save your answers in this PDF file.

2. **Prepare Local Environment**
   To prepare the local environment for the experiment, clone the case study documentation repository which contains all necessary artifacts. For the development, Open Policy Agent is required. Therefore, install OPA to your local machine. Additionally, install the OPA plugin for VS Code.

## 2 Development of Authorization Policies

The case study is performed on an excerpt of the BestRentalAppV2.0. The use cases "Add Car to Fleet", "Rent a Car", "List Customer Rentals", and "List Car Rentals" are used to create the required authorization artifacts.

To create the artifacts, guidelines and best practices for the E-AuthZ approach are provided in the GitLab repository Guidelines and Best Practices for E-AuthZ.

4. **Authorization Requirements**
   Create the authorization requirements for the aforementioned use cases. In the documentation repository, for each use case, a dedicated markdown page containing the template for the authorization requirement has been created. Use the respective guideline "Derivation of an Authorization Requirement from a Use Case" to create the authorization requirements.

5. **Authorization Policies**
   Create the authorization policies for the previously created authorization requirements. Use the API specifications of the application microservices which are linked in the README.md file. In addition, example access tokens for a customer and a fleet manager are provided. Use the respective guideline "Derivation of Authorization Policies from Authorization Requirements" to create the authorization policies. The authorization requirements of the other use cases are linked in the file README.md file.

6. **Authorization Policy Implementation in Rego**
   Implement the authorization policies in Rego. An example input from Envoy can be found in the file "example_envoy_input.json". The policies should be implemented next to the existing authorization policies in the folder policies located in the documentation repository. Use the respective guideline "Implement Rego Policies" for the implementation. The data folder already contains functions to access the attributes from a Policy Information Point (PIP).

The use case "Add Car to Fleet" creates an interaction between the application microservices of BestRentalAppV2.0. These use case requires the development of a service-to-service authorization policy. If an API token is required, use the name of the microservice (e.g. "AM-FleetManagementV1.0").

7. **Identify Service-to-Service Calls**
   Identify the service-to-service calls for the use case and note them in the related markdown page. The guideline "Derivation of Service-to-Service Authorization Policies" is provided to identify the calls.

8. **Derive Service-to-Service Authorization Policies**
   Derive the authorization policies for the service-to-service calls. Document the authorization policies in the related markdown page. The guideline "Derivation of Service-to-Service Authorization Policies" is provided to derive the authorization policies.

9. **Implement Service-to-Service Authorization Policies**
   Implement the service-to-service authorization policies using the provided guideline "Implementation of Service-to-Service Authorization". The implementation should be performed next to the previously implemented Rego policies.

## 3 Conclusion

10. **Concluding Evaluation**
    Fill out the survey "Survey Part 2: Evaluation of the Approach" in Section 4.2. Save your answers in this PDF file.

11. **Send Case Study Artifacts**
    Create a ZIP archive of the local case study documentation folder, including this PDF. Upload the ZIP archive anonymously to bwSync&Share.

## 4 Survey

### 4.1 Part 1: General Questions

**What are you studying?**

Computer Science, B.Sc. Computer Science, M.Sc.

Information Science, B.Sc. Information Science, M. Sc.

Other

**Which semester are you in?**

1-2 3-4 5-6 7-8 9-10

Other

**How familiar are you with the development of microservice-based applications?**

Very unfamiliar Unfamiliar Soomewhat familiar Familiar Very familiar

**How familiar are you with Attribute Based Access Control (ABAC)**

Very unfamiliar Unfamiliar Soomewhat familiar Familiar Very familiar

**Have you implemented authorization in an application?**

Yes No

**If yes, describe the application and how authorization has been implemented:**

**How familiar are you with the Rego policy language?**

Very unfamiliar Unfamiliar Soomewhat familiar Familiar Very familiar

### 4.2 Part 2: Evaluation of the Approach

***Authorization Requirements***

**The creation of authorization requirements is easy.**
Strongly agree Agree Undecided Disagree Strongly disagree

**The provided guidelines support the creation of authorization requirements.**
Strongly agree Agree Undecided Disagree Strongly disagree

**Do you have suggestions for improvements with regards to authorization requirements?**

***Authorization Policies***

**The creation of authorization policies is easy.**
Strongly agree Agree Undecided Disagree Strongly disagree

**The provided guidelines support the creation of authorization policies.**
Strongly agree Agree Undecided Disagree Strongly disagree

**Do you have suggestions for improvements with regards to authorization policies?**

***Rego Policies***

**The implementation of Rego policies is easy.**
Strongly agree Agree Undecided Disagree Strongly disagree

**The provided guidelines support the implementation of Rego policies.**
Strongly agree Agree Undecided Disagree Strongly disagree

**Do you have suggestions for improvements with regards to the implementation of Rego policies?**

***Service-to-Service Authorization***

**The derivation of service-to-service authorization policies is easy.**
Strongly agree Agree Undecided Disagree Strongly disagree

**The provided guidelines support the derivation of service-to-service authorization policies.**
Strongly agree Agree Undecided Disagree Strongly disagree

**Do you have suggestions for improvements with regards to the creation of service-to-service authorization policies?**

***General Approach***

**How do you evaluate the systematic development of authorization using the E-AuthZ approach?**

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
|---|---|---|---|---|---|---|---|---|
| not understandable | | | | | | | | understandable |
| not complex | | | | | | | | complex |
| clear | | | | | | | | confusing |
| complicated | | | | | | | | easy |
| incomplete | | | | | | | | complete |
| supportive | | | | | | | | not supportive |
| high traceability[1] | | | | | | | | no traceability[1] |

**Do you have suggestions to improve the approach?**

[1] Traceability between authorization artifacts

7

### A.5.4 Additional Case Study Results

| Question | Mean | Median | SD |
|---|---|---|---|
| The creation of authorization requirements is easy. | 4.08 | 4.0 | 0.86 |
| The provided guidelines support the creation of authorization requirements. | 4.38 | 4.0 | 0.51 |
| The creation of authorization policies is easy. | 3.85 | 4.0 | 1.07 |
| The provided guidelines support the creation of authorization policies. | 4.08 | 4.0 | 1.12 |
| The implementation of Rego policies is easy. | 3.77 | 4.0 | 1.01 |
| The provided guidelines support the implementation of Rego policies. | 4.08 | 4.0 | 0.76 |

**Table A.2:** Results for Goal 1

| Question | Mean | Median | SD |
|---|---|---|---|
| The derivation of service-to-service authorization policies is easy. | 3.92 | 4.0 | 1.12 |
| The provided guidelines support the derivation of service-to-service authorization policies. | 4.00 | 4.0 | 1.08 |

**Table A.3:** Results for Goal 2

| Question (1 / 7) | Mean | Median | SD |
|---|---|---|---|
| Not Understandable / Understandable | 5.92 | 6.0 | 0.86 |
| Complex / Not Complex | 4.77 | 5.0 | 1.30 |
| Confusing / Clear | 5.00 | 5.0 | 1.53 |
| Complicated / Easy | 4.69 | 5.0 | 1.11 |
| Incomplete / Complete | 5.23 | 5.0 | 1.42 |
| Not Supportive / Supportive | 5.38 | 6.0 | 1.71 |
| No Traceability / High Traceability | 5.38 | 6.0 | 1.39 |

**Table A.4:** Results for Overall Feedback

# B List of Abbreviations

**ABAC** Attribute Based Access Control.

**ABNF** Augmented Backus Naur Form.

**ACL** Access Control List.

**ALFA** Abbreviated Language for Authorization.

**API** Application Programming Interface.

**BPEL** Business Process Execution Language.

**BPMN** Business Process Model and Notation.

**CA** Certificate Authority.

**CAP** Consistency Availability Partition tolerance.

**CI/CD** Continuous Integration / Continuous Deployment.

**CNCF** Cloud Native Computing Foundation.

**CRUD** Create Read Update Delete.

**DDD** Domain-Driven Design.

**DP** Digital Policy.

**GQM** Goal Question Metric.

**gRPC** gRPC Remote Procedure Calls.

**IAM** Identity and Access Management.

**JSON** JavaScript Object Notation.

**JWK** JSON Web Key.

**JWT** JSON Web Token.

**LLM** Large Language Model.

**MAF** Microservice Authorization Framework.

**mTLS** Mutual Transport Layer Security.

**NIST** National Institute of Standards and Technology.

**NLP** Natural Language Policy.

**OASIS** Organization for the Advancement of Structured Information Standards.

**OCI** Open Container Initiative.

**OIDC** OpenID Connect.

**OPA** Open Policy Agent.

**OWASP** Open Web Application Security Project.

**PAP** Policy Administration Point.

**PDP** Policy Decision Point.

**PEP** Policy Enforcement Point.

**PIP** Policy Information Point.

**RBAC** Role Based Access Control.

**REST** REpresentational State Transfer.

**RPC** Remote Procedure Call.

**S2S** Service-to-Service.

**SaaS** Software as a Service.

**SAML** Security Assertion Markup Language.

**SDLC** Software Development Life Cycle.

**SOA** Service-oriented Architecture.

**SoaML** Service-oriented Architecture Modeling Language.

**SSO** Single Sign-On.

**TLS** Transport Layer Security.

**UI** User Interface.

**UML** Unified Modeling Language.

**URL** Uniform Resource Locator.

**VIN** Vehicle Identification Number.

**XACML** eXtensible Access Control Markup Language.

**YAML** YAML Ain't Markup Language.

# C List of Figures

# D List of Tables

# E List of Listings

# F List of Publications

**[ST+21]** **Niklas Sänger**, Stefan Throner, Simon Hanselmann, Michael Schneider, Sebastian Abeck: A Developer Portal for DevOps Environment, International Conference on Software Engineering Advances (ICSEA), 2021.

**[TH+21]** Stefan Throner, Heiko Hütter, **Niklas Sänger**, Michael Schneider, Simon Hanselmann, Patrick Petrovic, Sebastian Abeck: An Advanced DevOps Environment for Microservice-based Applications, IEEE SOSE, 2021.

**[SA22]** **Niklas Sänger**, Sebastian Abeck: Authentication and Authorization in Microservice-Based Applications, INFORMATIK 2022, GI-Jahrestagung, Hamburg, 2022.

**[SA23]** **Niklas Sänger**, Sebastian Abeck: User Authorization in Microservice-Based Application Engineering, MDPI Journal Software, 2023.

**[SA24]** **Niklas Sänger**, Sebastian Abeck: Fine-Grained Authorization in Microservice Architecture: A Decentralized Approach, The 39th ACM/SIGAPP Symposium on Applied Computing, Avila, Spain, 2024.

**[SA24b]** **Niklas Sänger**, Sebastian Abeck: Externalized and Decentralized Authorization of Microservices, Accepted at 20th International Workshop on Engineering Service-Oriented Applications and Cloud Services (WESOACS) at the 22nd Internation Conference on Service-Oriented Computing (ICSOC), Tunis, Tunisia, 2024.

# G Bibliography


[AC22] Murilo Góes de Almeida and Edna Dias Canedo. “Authentication and Authorization in Microservices Architecture: A Systematic Literature Review”. en. In: *Applied Sciences* 12.6 (2022-03), p. 3023. ISSN: 2076-3417. DOI: 10.3390/app12063023.

[An72] James P. Anderson. *Computer Security Technology Planning Study*. en. Tech. rep. ESD-TR-73-51. 1972. URL: https://apps.dtic.mil/sti/citations/tr/AD0758206 (visited on 2023-01-26).

[AQ+18] Muhammad umar Aftab, Zhiguang Qin, Zakria, Safeer Ali, Pirah, and Jalaluddin Khan. “The Evaluation and Comparative Analysis of Role Based Access Control and Attribute Based Access Control Model”. In: *2018 15th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*. ISSN: 2576-8964. 2018-12, pp. 35–39. DOI: 10.1109/ICCWAMTIP.2018.8632578.

[AS+02] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. “Agile software development methods: Review and analysis”. In: *CoRR* abs/1709.08439 (2017). URL: http://arxiv.org/abs/1709.08439 (visited on 2024-07-22).

[AT+19] Manar Alohaly, Hassan Takabi, and Eduardo Blanco. “Automated extraction of attributes from natural language attribute-based access control (ABAC) Policies”. en. In: *Cybersecurity* 2.1 (2019-12), p. 2. ISSN: 2523-3246. DOI: 10.1186/s42400-018-0019-2.

[BA+90] Michael Burrows, Martin Abadi, and Roger Needham. “A Logic of Authentication”. In: *ACM Trans. Comput. Syst.* 8.1 (1990-02). Publisher: ACM New York, NY, USA, pp. 18–36. ISSN: 0734-2071. DOI: 10.1145/77648.77649.

[BC+94] Victor R Basili, Gianluigi Caldiera, and H Dieter Rombach. “THE GOAL QUESTION METRIC APPROACH”. en. In: *Encyclopedia of software engineering* (1994), pp. 528–532.

[BG+17] David Brossard, Gerry Gebel, and Mark Berg. “A Systematic Approach to Implementing ABAC”. en. In: *Proceedings of the 2nd ACM Workshop on Attribute-Based Access Control - ABAC ’17*. Scottsdale, Arizona, USA: ACM Press, 2017, pp. 53–59. ISBN: 978-1-4503-4910-9. DOI: 10.1145/3041048.3041051.

[BG+22] Davide Berardi, Saverio Giallorenzo, Jacopo Mauro, Andrea Melis, Fabrizio Montesi, and Marco Prandini. “Microservice security: a systematic literature review”. en. In: *PeerJ Computer Science* 7 (2022-01), e779. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.779.

[BJ-FEV] *Gesetz zur Förderung der elektronischen Verwaltung*. URL: http://www.gesetze-im-internet.de/egovg/ (visited on 2022-10-18).

[BK+12] Marianne Busch, Nora Koch, Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. “Towards model-driven development of access control policies for web applications”. en. In: *Proceedings of the Workshop on Model-Driven Security*. Innsbruck Austria: ACM, 2012-10, pp. 1–6. ISBN: 978-1-4503-1806-8. DOI: 10.1145/2422498.2422502.

[BK+18] A. Banati, E. Kail, K. Karoczkai, and M. Kozlovszky. “Authentication and authorization orchestrator for microservice-based software architectures”. In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. Opatija: IEEE, 2018-05, pp. 1180–1184. ISBN: 978-953-233-095-3. DOI: 10.23919/MIPRO.2018.8400214.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. “Implementing Remote Procedure Calls”. In: *ACM Trans. Comput. Syst.* 2.1 (1984-02), pp. 39–59. ISSN: 0734-2071. DOI: 10.1145/2080.357392.

[BO16] Brendan Burns and David Oppenheimer. “Design patterns for Container-based Distributed Systems”. In: *8th USENIX workshop on hot topics in cloud computing (HotCloud 16)*. Denver, CO: USENIX Association, 2016-06. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns.

[BO+21] Christoph Buck, Christian Olenberger, André Schweizer, Fabiane Völter, and Torsten Eymann. “Never trust, always verify: A multivocal literature review on current knowledge and research gaps of zero-trust”. en. In: *Computers & Security* 110 (2021-11), p. 102436. ISSN: 01674048. DOI: 10.1016/j.cose.2021.102436.

[CB+21] Ramaswamy Chandramouli, Zack Butcher, and Aradhna Chetal. *Attribute-based Access Control for Microservices-based Applications Using a Service Mesh*. en. Tech. rep. National Institute of Standards and Technology, 2021-08. DOI: 10.6028/NIST.SP.800-204B.

[CD+14] Craigen, D., Diakun-Thibault, N., and Purse, R. “Defining Cybersecurity”. en. In: *Defining Cybersecurity. Technology Innovation Management Review* 4 (2014), pp. 13–21. DOI: 10.22215/timreview/835.

[Ch19] Ramaswamy Chandramouli. *Security Strategies for Microservices-based Application Systems*. en. Tech. rep. NIST SP 800-204. Gaithersburg, MD: National Institute of Standards and Technology, 2019-08, NIST SP 800–204. DOI: 10.6028/NIST.SP.800-204.

[CN-AS] Cloud Native Computing Foundation. *CNCF Annual Survey 2023*. en-US. 2024-04. URL: https://www.cncf.io/reports/cncf-annual-survey-2023/ (visited on 2024-12-06).

[CO-Doc] Casbin Organization. *Documentation: Overview*. en. 2024-07. URL: https://casbin.org/docs/overview (visited on 2024-06-27).

[Co00] Alistair Cockburn. *Writing Effective Use Cases*. 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0-201-70225-8.

[Co05] Alistair Cockburn. *The Pattern: Ports and Adapters ("Object Structural")*. 2005. URL: https://alistair.cockburn.us/hexagonal-architecture/ (visited on 2024-11-07).

[CO08] D. Crocker and P. Overell. *Augmented BNF for Syntax Specifications: ABNF*. STD 68. RFC Editor, 2008-01. DOI: https://doi.org/10.17487/RFC5234.

[CW+18] Carlos Cotrini, Thilo Weghorn, and David Basin. "Mining ABAC Rules from Sparse Logs". In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018-04, pp. 31–46. DOI: 10.1109/EuroSP.2018.00011.

[DB13] Jeffrey Dean and Luiz André Barroso. "The tail at scale". en. In: *Communications of the ACM* 56.2 (2013-02), pp. 74–80. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2408776.2408794.

[De17] Brajesh De. *API Management*. en. Berkeley, CA: Apress, 2017. ISBN: 978-1-4842-1306-3 978-1-4842-1305-6. DOI: 10.1007/978-1-4842-1305-6.

[DG+17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. "Microservices: Yesterday, Today, and Tomorrow". en. In: *Present and Ulterior Software Engineering*. Ed. by Manuel Mazzara and Bertrand Meyer. Cham: Springer International Publishing, 2017, pp. 195–216. ISBN: 978-3-319-67424-7 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_12.

[DM+18] Saptarshi Das, Barsha Mitra, Vijayalakshmi Atluri, Jaideep Vaidya, and Shamik Sural. "Policy Engineering in RBAC and ABAC". en. In: *From Database to Cyber Security*. Ed. by Pierangela Samarati, Indrajit Ray, and Indrakshi Ray. Vol. 11170. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 24–54. ISBN: 978-3-030-04833-4 978-3-030-04834-1. DOI: 10.1007/978-3-030-04834-1_2.

[Do-Do] Docker Inc. *Docker: Accelerated Container Application Development*. en-US. 2022-05. URL: https://www.docker.com/ (visited on 2024-11-07).

[DP06] Wolfgang Dobmeier and Günther Pernul. "Modellierung von Zugriffsrichtlinien für offene Systeme". In: *EMISA 2006 – Methoden, Konzepte und Technologien für die Entwicklung von dienstbasierten Informationssystemen – Beiträge des Workshops der GI-Fachgruppe EMISA (Entwicklungsmethoden für Informationssystemeund deren Anwendung)*. Bonn: Gesellschaft für Informatik e. V., 2006, pp. 35–47. ISBN: 978-3-88579-189-8.

[Du16] Zoya Durdik. *Architectural Design Decision Documentation through Reuse of Design Patterns*. Karlsruhe: KIT Scientific Publishing, 2016-07. ISBN: 978-3-7315-0292-0. DOI: 10.5445/KSP/1000043807.

[DWP23] Deparment for Work & Pensions. *Security Standard – Microservices Architecture (SS-028)*. Tech. rep. 2023-11. URL: https://assets.publishing.service.gov.uk/media/65788847254aaa0010050b88/dwp-ss-028-security-standard-microservices-architecture-v2.pdf (visited on 2023-11-21).

[EB+11] Brian Elvesæter, Arne-Jørgen Berre, and Andrey Sadovykh. "SPECIFYING SERVICES USING THE SERVICE ORIENTED ARCHITECTURE MODELING LANGUAGE (SOAML) - A baseline for Specification of Cloud-based Services:" en. In: *Proceedings of the 1st International Conference on Cloud Computing and Services Science*. Noordwijkerhout, Netherlands: SciTePress - Science, and Technology Publications, 2011, pp. 276–285. ISBN: 978-989-8425-52-2. DOI: 10.5220/0003393202760285.

[EK10] Aaron Elliott and Scott Knight. "Role Explosion: Acknowledging the Problem". en. In: *Software Engineering research and practice*. 2010, pp. 349–355.

[EP-Doc] Envoy Project. *Envoy Documentation: What is Envoy?* 2023-04. URL: https://www.envoyproxy.io/docs/envoy/v1.26.0/intro/what_is_envoy (visited on 2023-04-24).

[EP-EA] Envoy Project. *Envoy Documentation: HTTP filters - External Authorization*. URL: https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/ext_authz_filter (visited on 2023-03-29).

[Er18] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Number: 8446. 2018-08. DOI: 10.17487/RFC8446.

[Ev03] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. en. Boston: Addison-Wesley, 2004. ISBN: 978-0-321-12521-7.

[FF+18] Andrei Furda, Colin Fidge, Olaf Zimmermann, Wayne Kelly, and Alistair Barros. "Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency". In: *IEEE Software* 35.3 (2018-05), pp. 63–72. ISSN: 1937-4194. DOI: 10.1109/MS.2017.440134612.

[Fi00] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". en. PhD thesis. UNIVERSITY OF CALIFORNIA, IRVINE, 2000. URL: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm (visited on 2023-03-02).

[Fi03] Donald Firesmith. "Engineering Security Requirements." en. In: *The Journal of Object Technology* 2.1 (2003), p. 53. ISSN: 1660-1769. DOI: 10.5381/jot.2003.2.1.c6.

[FL14] Martin Fowler and James Lewis. *Microservices*. URL: https://martinfowler.com/articles/microservices.html (visited on 2024-10-15).

[FM+17] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption". In: *2017 IEEE International Conference on Software Architecture (ICSA)*. 2017-04, pp. 21–30. DOI: 10.1109/ICSA.2017.24.

[FP-Wha] Fluentd Project. *What is Fluentd? | Fluentd*. en. URL: https://www.fluentd.org/architecture (visited on 2024-07-01).

[FS-Tra] FudanSELab. *FudanSELab/train-ticket*. 2024-10. URL: https://github.com/FudanSELab/train-ticket (visited on 2024-10-21).

[FV04] Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno. "An Introduction to UML Profiles". en. In: *UML and Model Engineering* 2 (2004-02), pp. 6–13. ISSN: 1684-5285.

[FZ+21] Athareh Fatemian, Bahman Zamani, Marzieh Masoumi, Mehran Kamranpour, Behrouz Tork Ladani, and Shekoufeh Kolahdouz Rahimi. "Automatic Generation of XACML Code using Model-Driven Approach". en. In: *2021 11th International Conference on Computer Engineering and Knowledge (ICCKE)*. Mashhad, Iran, Islamic Republic of: IEEE, 2021-10, pp. 206–211. ISBN: 978-1-66540-208-8. DOI: 10.1109/ICCKE54056.2021.9721518.

[GA-Do] gRPC Authors. *Documentation | gRPC*. en. URL: https://grpc.io/docs/ (visited on 2024-10-27).

[GF13] Seyed Hossein Ghotbi and Bernd Fischer. “Fine-Grained Role- and Attribute-Based Access Control for Web Applications”. en. In: *Software and Data Technologies*. Ed. by José Cordeiro, Slimane Hammoudi, and Marten van Sinderen. Vol. 411. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 171–187. ISBN: 978-3-642-45403-5. DOI: 10.1007/978-3-642-45404-2_12.

[GF94] O.C.Z. Gotel and C.W. Finkelstein. “An analysis of the requirements traceability problem”. In: *Proceedings of IEEE International Conference on Requirements Engineering*. 1994-04, pp. 94–101. DOI: 10.1109/ICRE.1994.292398.

[Gi18] Pascal Giessler. “Domänengetriebener Entwurf von ressourcenorientierten Microservices”. German. PhD thesis. Karlsruher Institut für Technologie (KIT), 2018. DOI: 10.5445/IR/1000083352.

[GL-k6] Grafana Labs. *Load testing for engineering teams | Grafana k6*. en. URL: https://k6.io (visited on 2024-10-17).

[GL02] Seth Gilbert and Nancy Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. en. In: *ACM SIGACT News* 33.2 (2002-06), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601.

[Gl07] M. Glinz. “On Non-Functional Requirements”. en. In: *15th IEEE International Requirements Engineering Conference (RE 2007)*. Delhi: IEEE, 2007-10, pp. 21–26. ISBN: 978-0-7695-2935-6. DOI: 10.1109/RE.2007.45.

[GL+18] Manuel Gotin, Felix Lösch, Robert Heinrich, and Ralf Reussner. “Investigating Performance Metrics for Scaling Microservices in CloudIoT-Environments”. en. In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. Berlin Germany: ACM, 2018-03, pp. 157–167. ISBN: 978-1-4503-5095-2. DOI: 10.1145/3184407.3184430.

[Go-Con] Go. *context*. 2024. URL: https://pkg.go.dev/context (visited on 2024-07-28).

[Go-Pro] Google LLC All. *Protocol Buffers Documentation*. en-US. 2023-03. URL: https://protobuf.dev/programming-guides/proto3/ (visited on 2023-03-16).

[Go10] Dieter Gollmann. “Computer security”. en. In: *WIREs Computational Statistics* 2.5 (2010), pp. 544–554. ISSN: 1939-0068. DOI: 10.1002/wics.106.

[GP+06] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. “Attribute-based encryption for fine-grained access control of encrypted data”. en. In: *Proceedings of the 13th ACM conference on Computer and communications security*. Alexandria Virginia USA: ACM, 2006-10, pp. 89–98. ISBN: 978-1-59593-518-2. DOI: 10.1145/1180405.1180418.

[GS87] Hector Garcia-Molina and Kenneth Salem. “Sagas”. In: *SIGMOD Rec.* 16.3 (1987), pp. 249–259. ISSN: 0163-5808. DOI: 10.1145/38714.38742.

[Gu02] Michele D. Guel. “A Framework for Choosing Your Next Generation Authentication/Authorization System”. en. In: *Information Security Technical Report* 7.1 (2002-03), pp. 63–78. ISSN: 13634127. DOI: 10.1016/S1363-4127(02)00107-3.

[GZ+21] Weize Guo, Li Zhang, and Xiaoli Lian. *Automatically detecting the conflicts between software requirements based on finer semantic analysis*. en. arXiv:2103.02255 [cs]. 2021-03. URL: http://arxiv.org/abs/2103.02255 (visited on 2023-12-08).

[Ha12] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. ISSN: 2070-1721. RFC Editor, 2012-10. DOI: 10.17487/RFC6749.

[Ha88] Norm Hardy. “The Confused Deputy: (or why capabilities might have been invented)”. en. In: *ACM SIGOPS Operating Systems Review* 22.4 (1988-10), pp. 36–38. ISSN: 0163-5980. DOI: 10.1145/54289.871709.

[He21] Gal Helemski. *How PlainID Solves the OPA Manageability Gap*. en. URL: https://blog.plainid.com/plainid-solves-opa-manageability-gap (visited on 2024-07-01).

[HF+14] Vincent C. Hu, David Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. en. Tech. rep. NIST SP 800-162. National Institute of Standards and Technology, 2014-01, NIST SP 800–162. DOI: 10.6028/NIST.SP.800-162.

[HG+17] Benjamin Hippchen, Pascal Giessler, Roland Steinegger, Michael Schneider, and Sebastian Abeck. “Designing Microservice-Based Applications by Using a Domain-Driven Design Approach”. In: *International Journal on Advances in Software* 10.3&4 (2017), pp. 432–445.

[HK+17] Vincent C Hu, Rick Kuhn, and Dylan Yaga. *Verification and Test Methods for Access Control Policies/Models*. en. Tech. rep. NIST SP 800-192. Gaithersburg, MD: National Institute of Standards and Technology, 2017-06, NIST SP 800–192. DOI: 10.6028/NIST.SP.800-192.

[HK+21] John Heaps, Ram Krishnan, Yufei Huang, Jianwei Niu, and Ravi Sandhu. “Access Control Policy Generation from User Stories Using Machine Learning”. en. In: *Data and Applications Security and Privacy XXXV*. Ed. by Ken Barker and Kambiz Ghazinour. Vol. 12840. Series Title: Lecture Notes in Computer Science. Cham:

Springer International Publishing, 2021, pp. 171–188. ISBN: 978-3-030-81241-6 978-3-030-81242-3. DOI: 10.1007/978-3-030-81242-3_10.

[HM05] John Hughes and Eve Maler. *Security Assertion Markup Language (SAML) V2.0 Technical Overview*. Tech. rep. Publisher: Citeseer. OASIS Security Services TC, 2008-03, p. 12. URL: http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html (visited on 2023-01-27).

[HO22] Dpa. *Ein Jahr nach dem Erpressungstrojaner: Anhalt-bitterfeld spürt noch die folgen*. 2022-07. URL: https://heise.de/-7162431 (visited on 2022-10-20).

[HP+99] Russ Housley, Tim Polk, Dr. Warwick S. Ford, and Dave Solo. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*. Number: 2459 Series: Request for comments. 1999-01. DOI: 10.17487/RFC2459.

[IA-Do] Istio Authors. *Istio / Documentation*. en. URL: https://istio.io/latest/docs/ (visited on 2024-07-16).

[IA+18] I. Indu, P.M. Rubesh Anand, and Vidhyacharan Bhaskar. "Identity and access management in cloud environment: Mechanisms and challenges". en. In: *Engineering Science and Technology, an International Journal* 21.4 (2018-08), pp. 574–588. ISSN: 22150986. DOI: 10.1016/j.jestch.2018.05.010.

[ISO-247] *ISO/IEC/IEEE 24765:2017 Systems and software engineering — Vocabulary*. URL: https://www.iso.org/standard/71952.html (visited on 2024-10-25).

[JJ+19] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. "Performance Modeling for Cloud Microservice Applications". en. In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. Mumbai India: ACM, 2019-04, pp. 25–32. ISBN: 978-1-4503-6239-9. DOI: 10.1145/3297663.3310309.

[JS+12] Xin Jin, Ravi Sandhu, and Ram Krishnan. "RABAC: Role-Centric Attribute-Based Access Control". en. In: *Computer Network Security*. Ed. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Igor Kotenko, and Victor Skormin. Vol. 7531. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 84–96. ISBN: 978-3-642-33703-1 978-3-642-33704-8. DOI: 10.1007/978-3-642-33704-8_8.

[JS+18] Brenda Jin, Saurabh Sahni, and Amir Shevat. *Designing Web APIs: Building APIs that Developers Love*. eng. First edition. Beijing Boston Farnham Sebastopol Tokio: O'Reilly, 2018. ISBN: 978-1-4920-2692-1 978-1-4920-2687-7.

[Ju06] Matjaz B. Juric. *A Hands-on Introduction to BPEL*. URL: https://www.oracle.com/technical-resources/articles/matjaz-bpel.html (visited on 2024-09-30).

[KA-Doc] Keycloak Authors. *Documentation - Keycloak*. URL: https://www.keycloak.org/documentation (visited on 2024-11-07).

[Ki15] Joseph Migga Kizza. "Access Control and Authorization". en. In: *Guide to Computer Network Security*. Series Title: Computer Communications and Networks. London: Springer London, 2015, pp. 185–204. ISBN: 978-1-4471-6653-5 978-1-4471-6654-2. DOI: 10.1007/978-1-4471-6654-2_9.

[KK+21] Rafiq Ahmad Khan, Siffat Ullah Khan, Habib Ullah Khan, and Muhammad Ilyas. "Systematic Mapping Study on Security Approaches in Secure Software Engineering". In: *IEEE Access* 9 (2021). Conference Name: IEEE Access, pp. 19139–19160. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3052311.

[Kl17] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. eng. First edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2017. ISBN: 978-1-4919-0310-0 978-1-4919-0311-7.

[Ko08] Heiko Koziolek. "Parameter dependencies for reusable performance specifications of software components". PhD thesis. Universitätsverlag Karlsruhe, 2008. DOI: 10.5445/KSP/1000009096.

[Ko08a] Heiko Koziolek. "Goal, Question, Metric". en. In: *Dependability Metrics*. Ed. by Irene Eusgeld, Felix C. Freiling, and Ralf Reussner. Vol. 4909. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 39–42. ISBN: 978-3-540-68946-1 978-3-540-68947-8. DOI: 10.1007/978-3-540-68947-8_6.

[KP+13] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. "On the Security of the TLS Protocol: A Systematic Analysis". en. In: *Advances in Cryptology – CRYPTO 2013*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8042. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 429–448. ISBN: 978-3-642-40040-7 978-3-642-40041-4. DOI: 10.1007/978-3-642-40041-4_24.

[KQ17] Nane Kratzke and Peter-Christian Quint. "Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study". en. In: *Journal of Systems and Software* 126 (2017-04), pp. 1–16. ISSN: 01641212. DOI: 10.1016/j.jss.2017.01.001.

[Ku-Doc] Kubernetes Authors. *Kubernetes Documentation*. en. URL: https://kubernetes.io/docs/home/ (visited on 2024-07-01).

[LB+02] Torsten Lodderstedt, David Basin, and Jürgen Doser. "SecureUML: A UML-Based Modeling Language for Model-Driven Security". en. In: *UML 2002 — The Unified Modeling Language*. Ed. by Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook. Vol. 2460. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 426–441. ISBN: 978-3-540-44254-7 978-3-540-45800-5. DOI: 10.1007/3-540-45800-X_33.

[LC+21] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen. "Automatic Policy Generation for Inter-Service Access Control of Microservices". en. In: USENIX Association, 2021, pp. 3971–3988. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/li-xing (visited on 2023-11-16).

[LF+09] Bo Lang, Ian Foster, Frank Siebenlist, Rachana Ananthakrishnan, and Tim Freeman. "A Flexible Attribute Based Access Control Method for Grid Computing". en. In: *Journal of Grid Computing* 7.2 (2009-06), pp. 169–180. ISSN: 1570-7873, 1572-9184. DOI: 10.1007/s10723-008-9112-1.

[LK+24] Yishu Li, Jacky Keung, Xiaoxue Ma, Chun Yong Chong, Jingyu Zhang, and Yihan Liao. "LLM-Based Class Diagram Derivation from User Stories with Chain-of-Thought Promptings". In: *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. ISSN: 2836-3795. 2024-07, pp. 45–50. DOI: 10.1109/COMPSAC61105.2024.00017.

[Lu14] Hongqian Karen Lu. "Keeping Your API Keys in a Safe". In: *2014 IEEE 7th International Conference on Cloud Computing*. ISSN: 2159-6190. 2014-06, pp. 962–965. DOI: 10.1109/CLOUD.2014.143.

[Ma12] Robert C. Martin. *The Clean Architecture*. 2012-08. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html (visited on 2024-10-19).

[MC+22] Airy Magnien, Gabriele Cecchetti, Anna Lina Ruscelli, Paul Hyde, Jin Liu, and Stefan Wegele. "Formalization and Processing of Data Requirements for the Development of Next Generation Railway Traffic Management Systems". en. In: *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*. Vol. 13294. Cham: Springer International Publishing, 2022, pp. 35–45. ISBN: 978-3-031-05813-4 978-3-031-05814-1. DOI: 10.1007/978-3-031-05814-1_3.

[Me02] D.A. Menasce. “Load testing of Web sites”. In: *IEEE Internet Computing* 6.4 (2002-07). Conference Name: IEEE Internet Computing, pp. 70–74. ISSN: 1941-0131. DOI: 10.1109/MIC.2002.1020328.

[MF18] Jonathan Mace and Rodrigo Fonseca. “Universal context propagation for distributed system instrumentation”. en. In: *Proceedings of the Thirteenth EuroSys Conference*. Porto Portugal: ACM, 2018-04, pp. 1–18. ISBN: 978-1-4503-5584-1. DOI: 10.1145/3190508.3190526.

[MH+23] Catherine Meadows, Sena Hounsinou, Timothy Wood, and Gedare Bloom. “Sidecar-based Path-aware Security for Microservices”. en. In: *Proceedings of the 28th ACM Symposium on Access Control Models and Technologies*. Trento Italy: ACM, 2023-05, pp. 157–162. ISBN: 9798400701733. DOI: 10.1145/3589608.3594742.

[MM+21] Loic Miller, Pascal Merindol, Antoine Gallais, and Cristel Pelsser. “Towards Secure and Leak-Free Workflows Using Microservice Isolation”. en. In: *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*. Paris, France: IEEE, 2021-06, pp. 1–5. ISBN: 978-1-66544-005-9. DOI: 10.1109/HPSR52026.2021.9481820.

[MM+24] Fabio Martinelli, Francesco Mercaldo, Luca Petrillo, and Antonella Santone. “Security Policy Generation and Verification through Large Language Models: A Proposal”. en. In: *Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy*. Porto Portugal: ACM, 2024-06, pp. 143–145. ISBN: 9798400704215. DOI: 10.1145/3626232.3658635.

[Ne15] Sam Newman. *Building microservices: designing fine-grained systems*. en. First Edition. Beijing Sebastopol, CA: O’Reilly Media, 2015. ISBN: 978-1-4919-5035-7.

[Ne19] Sam Newman. *Monolith to microservices: evolutionary patterns to transform your monolith*. eng. First edition. Beijing Boston Farnham Sebastopol Tokyo: O’Reilly, 2019. ISBN: 978-1-4920-4784-1.

[Net-Bui] Netflix Technology Blog. *Building Netflix’s Distributed Tracing Infrastructure*. Tech. rep. 2020. URL: https://netflixtechblog.com/building-netflixs-distributed-tracing-infrastructure-bb856c319304 (visited on 2024-10-02).

[NJ+18] Antonio Nehme, Vitor Jesus, Khaled Mahbub, and Ali Abdallah. “Fine-Grained Access Control for Microservices”. en. In: *Foundations and Practice of Security*. Ed. by Nur Zincir-Heywood, Guillaume Bonfante, Mourad Debbabi, and Joaquin Garcia-Alfaro. Vol. 11358. Cham: Springer International Publishing, 2019, pp. 285–300. ISBN: 978-3-030-18418-6 978-3-030-18419-3. DOI: 10.1007/978-3-030-18419-3_19.

[NK+17] Masoud Narouei, Hamed Khanpour, Hassan Takabi, Natalie Parde, and Rodney Nielsen. “Towards a Top-down Policy Engineering Framework for Attribute-based Access Control”. en. In: *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*. Indianapolis Indiana USA: ACM, 2017-06, pp. 103–114. ISBN: 978-1-4503-4702-0. DOI: 10.1145/3078861.3078874.

[NY+05] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. *The Kerberos Network Authentication Service (V5)*. RFC 4120. ISSN: 2070-1721. RFC Editor, 2005-07. DOI: 10.17487/RFC4120.

[OAS-ALF] *ALFA Language Basics*. URL: https://alfa.guide/alfa-authorization-language/ (visited on 2024-08-21).

[OAS-XAC] OASIS Open. *eXtensible Access Control Markup Language (XACML) Version 3.0*. en. Tech. rep. OASIS Open, 2013-01, p. 154. URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-en.html (visited on 2022-05-10).

[OMG-UML] Object Management Group. “Unified Modeling Language, v2.5.1”. en. In: *Unified Modeling Language* (2017-12), p. 796. URL: https://www.omg.org/spec/UML/2.5.1/PDF (visited on 2022-10-18).

[OP-Do] Open Policy Agent contributors. *Open Policy Agent | Documentation*. en-US. 2023-03. URL: https://www.openpolicyagent.org/docs/latest/ (visited on 2023-03-16).

[Ope-Doc] OpenTelemetry. *Documentation*. Tech. rep. 2024. URL: https://opentelemetry.io/docs/ (visited on 2024-08-08).

[Ope-Spe] Open API Initiative. *Open API Specification - v3.1.0*. en-US. Tech. rep. 2023-03. URL: https://spec.openapis.org/oas/v3.1.0 (visited on 2023-03-16).

[OW21] OWASP Foundation. *OWASP Top 10:2021*. Tech. rep. 2021. URL: https://owasp.org/Top10/ (visited on 2022-02-15).

[OW21a] OWASP Foundation. *OWASP Application Security Verification Standard (ASVS)*. 2021-10. URL: https://raw.githubusercontent.com/OWASP/ASVS/v4.0.3/4.0/OWASP%20Application%20Security%20Verification%20Standard%204.0.3-en.pdf (visited on 2024-09-11).

[PB68] *SOFTWARE ENGINEERING*. en. Tech. rep. Garmisch, Germany, 1968.

[Pe-OP] Permit.io. *Introduction to OPAL*. en. URL: https://docs.opal.ac/getting-started/intro (visited on 2024-11-07).

[PG24] PostgreSQL Global Development Group. *PostgreSQL*. en. 2024-11. URL: https://www.postgresql.org/ (visited on 2024-11-19).

[Pi23] Roman Pilipchuk. *Architectural alignment of access control requirements extracted from business processes*. eng. The Karlsruhe series on software design and quality 35. Karlsruhe: KIT Scientific Publishing, 2023. ISBN: 978-3-7315-1212-7. DOI: 10.5445/IR/1000140856.

[PM+09] Anil Patel, Malcolm McRoberts, and Melissa Crenshaw. "Identity propagation in N-tier systems". en. In: *MILCOM 2009 - 2009 IEEE Military Communications Conference*. Boston, MA, USA: IEEE, 2009-10, pp. 1–5. ISBN: 978-1-4244-5238-5. DOI: 10.1109/MILCOM.2009.5379926.

[PM+12] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. eng. Hoboken, NJ: Wiley, 2012. ISBN: 978-1-118-10435-4. DOI: 10.1002/9781118181034.

[PR15] Klaus Pohl and Chris Rupp. *Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam, Foundation Level, IREB Compliant*. en. Second edition. Santa Barbara, CA: Rocky Nook, 2015. ISBN: 978-1-937538-77-4.

[Pr24] Sergey Pronin. *The inevitable Kubernetes - 10 years, still a lot to do*. en-US. 2024-07. URL: https://www.cncf.io/blog/2024/07/26/the-inevitable-kubernetes-10-years-still-a-lot-to-do/ (visited on 2024-12-06).

[PS+21] Francisco Ponce, Jacopo Soldani, Hernán Astudillo, and Antonio Brogi. *Smells and Refactorings for Microservices Security: A Multivocal Literature Review*. en. arXiv:2104.13303 [cs]. 2021-04. URL: http://arxiv.org/abs/2104.13303 (visited on 2023-08-16).

[PS+22] Francisco Ponce, Jacopo Soldani, Hernán Astudillo, and Antonio Brogi. "Should Microservice Security Smells Stay or be Refactored? Towards a Trade-off Analysis". en. In: *Software Architecture*. Ed. by Ilias Gerostathopoulos, Grace Lewis, Thais Batista, and Tomáš Bureš. Vol. 13444. Cham: Springer International Publishing, 2022, pp. 131–139. ISBN: 978-3-031-16696-9 978-3-031-16697-6. DOI: 10.1007/978-3-031-16697-6_9.

[RB+20] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. *Zero Trust Architecture*. Tech. rep. 800-207. National Institute of Standards and Technology, 2020-08. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf (visited on 2024-11-22).

[Ro70] Winston W Rovce. "Managing the development of large software systems: Concepts and techniques". en. In: *Proceedings of IEEE WESCON*. Vol. 26, pp. 328–388.

[Ru10] Nayan B. Ruparelia. “Software development lifecycle models”. en. In: *ACM SIGSOFT Software Engineering Notes* 35.3 (2010-05), pp. 8–13. ISSN: 0163-5948. DOI: 10.1145/1764810.1764814.

[Ru18] Chaitanya K. Rudrabhatla. “Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture”. en. In: *International Journal of Advanced Computer Science and Applications* 9.8 (2018). ISSN: 21565570, 2158107X. DOI: 10.14569/IJACSA.2018.090804.

[SA22] Niklas Sänger and Sebastian Abeck. “Authentication and Authorization in Microservice-Based Applications”. en. In: *INFORMATIK 2022 - Informatik in den Naturwissenschaften*. ISBN: 9783885797203. Gesellschaft für Informatik, Bonn, 2022, pp. 207–218. ISBN: 978-3-88579-720-3. DOI: 10.18420/INF2022_19.

[SA23] Niklas Sänger and Sebastian Abeck. “User Authorization in Microservice-Based Applications”. en. In: *Software* 2.3 (2023-09). Number: 3 Publisher: Multidisciplinary Digital Publishing Institute, pp. 400–426. ISSN: 2674-113X. DOI: 10.3390/software2030019.

[SA24] Niklas Sänger and Sebastian Abeck. “Fine-Grained Authorization in Microservice Architecture: A Decentralized Approach”. en. In: *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*. Avila Spain: ACM, 2024-04, pp. 1219–1222. ISBN: 9798400702433. DOI: 10.1145/3605098.3636121.

[SB+14] Natsuhiko Sakimura, John Bradley, Mike Jones, Breno De Medeiros, and Chuck Mortimore. “Openid connect core 1.0”. In: *The OpenID Foundation* (2014), S3. URL: https://openid.net/specs/openid-connect-core-1_0.html (visited on 2023-01-27).

[SB+23] N. Sakimura, J. Bradley, M. Joney, and E. Jay. *OpenID Connect Discovery 1.0 incorporating errata set 2*. 2023-12. URL: https://openid.net/specs/openid-connect-discovery-1_0.html (visited on 2024-11-19).

[Sc24] Michael Schneider. “Domänengetriebene Entwicklung von fortgeschrittenen Web-Anwendungen”. German. PhD thesis. Karlsruher Institut für Technologie (KIT), 2024. DOI: 10.5445/IR/1000169494.

[SC+96] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. “Role-based access control models”. en. In: *Computer* 29.2 (1996-02), pp. 38–47. ISSN: 00189162. DOI: 10.1109/2.485845.

[Se06] J. Sermersheim. *Lightweight Directory Access Protocol (LDAP): The Protocol*. RFC 4511. RFC Editor, 2006-06. DOI: 10.17487/RFC4511.

[SF+02] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. “Security Patterns”. en. In: *Informatik-Spektrum* 25.3 (2002-06), pp. 220–223. ISSN: 0170-6012, 1432-122X. DOI: 10.1007/s002870200223.

[SH+21] Martijn Sauwens, Emad Heydari Beni, Kristof Jannes, Bert Lagaisse, and Wouter Joosen. “ThunQ: A Distributed and Deep Authorization Middleware for Early and Lazy Policy Enforcement in Microservice Applications”. en. In: *Service-Oriented Computing*. Ed. by Hakim Hacid, Odej Kao, Massimo Mecella, Naouel Moha, and Hye-young Paik. Vol. 13121. Cham: Springer International Publishing, 2021, pp. 204–220. ISBN: 978-3-030-91430-1 978-3-030-91431-8. DOI: 10.1007/978-3-030-91431-8_13.

[SL20] Mike Swoyer and Loukides, Steve. *Microservices Adoption in 2020*. en-US. 2020-07. URL: https://www.oreilly.com/radar/microservices-adoption-in-2020/ (visited on 2023-06-20).

[Sm15] John Ferguson Smart. *BDD in action: Behavior-Driven Development for the whole software lifecycle*. en. Shelter Island, NY: Manning Publications, 2015. ISBN: 978-1-61729-165-4.

[SO15] Daniel Servos and Sylvia L. Osborn. “HGABAC: Towards a Formal Model of Hierarchical Attribute-Based Access Control”. en. In: *Foundations and Practice of Security*. Ed. by Frédéric Cuppens, Joaquin Garcia-Alfaro, Nur Zincir Heywood, and Philip W. L. Fong. Vol. 8930. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 187–204. ISBN: 978-3-319-17039-8 978-3-319-17040-4. DOI: 10.1007/978-3-319-17040-4_12.

[SO17] Daniel Servos and Sylvia L. Osborn. “Current Research and Open Problems in Attribute-Based Access Control”. en. In: *ACM Computing Surveys* 49.4 (2017-12), pp. 1–45. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3007204.

[Sol-Mic] *Microservices, Kubernetes and Istio – 2022 Adoption Trends*. en-US. URL: https://www.solo.io/resources/infographic/microservices-kubernetes-and-istio-2022-adoption-trends/ (visited on 2023-08-25).

[Sp-Aut] Spring by VMWare. *Authorization :: Spring Security*. URL: https://docs.spring.io/spring-security/reference/servlet/authorization/index.html (visited on 2023-09-21).

[SP17] Vindeep Singh and Sateesh K Peddoju. “Container-based microservice architecture for cloud applications”. In: *2017 International Conference on Computing, Communication and Automation (ICCCA)*. 2017-05, pp. 847–852. DOI: 10.1109/CCAA.2017.8229914.

[SR22] Statista Research Department. *Nutzung von Cloud Computing in Deutschen Unternehmen Bis 2020*. 2022-07. URL: https://de.statista.com/statistik/daten/studie/177484/umfrage/einsatz-von-cloud-computing-in-deutschen-unternehmen-2011/ (visited on 2022-10-18).

[SS+09] Jian Shu, Lianghong Shi, Bing Xia, and Linlan Liu. "Study on Action and Attribute-Based Access Control Model for Web Services". In: *2009 Second International Symposium on Information Science and Engineering*. ISSN: 2160-1291. 2009-12, pp. 213–216. DOI: 10.1109/ISISE.2009.80.

[SS+15] Martina Seidl, Marion Scholz, Christian Huemer, and Gerti Kappel. *UML @ Classroom*. en. Undergraduate Topics in Computer Science. Cham: Springer International Publishing, 2015. ISBN: 978-3-319-12741-5 978-3-319-12742-2. DOI: 10.1007/978-3-319-12742-2.

[SS94] R.S. Sandhu and P. Samarati. "Access control: principle and practice". In: *IEEE Communications Magazine* 32.9 (1994-09). Conference Name: IEEE Communications Magazine, pp. 40–48. ISSN: 1558-1896. DOI: 10.1109/35.312842.

[ST+21] Niklas Sänger, Stefan Throner, Simon Hanselmann, Michael Schneider, and Sebastian Abeck. "A Developer Portal for DevOps Environment". en. In: 2021, pp. 121–127. ISBN: 978-1-61208-894-5. URL: https://www.thinkmind.org/articles/icsea_2021_2_130_10078.pdf (visited on 2022-10-20).

[Sty24] Styra Inc. *Styra Documentation: Rego Style Guide*. en. URL: https://docs.styra.com/opa/rego-style-guide (visited on 2024-10-01).

[SV08] Michel Dos Santos Soares and Jos Vrancken. "Model-Driven User Requirements Specification using SysML". en. In: *Journal of Software* 3.6 (2008-06), pp. 57–68. ISSN: 1796-217X. DOI: 10.4304/jsw.3.6.57-68.

[TB+17] Tanay Talukdar, Gunjan Batra, Jaideep Vaidya, Vijayalakshmi Atluri, and Shamik Sural. "Efficient Bottom-Up Mining of Attribute Based Access Control Policies". In: *2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC)*. 2017-10, pp. 339–348. DOI: 10.1109/CIC.2017.00051.

[TD+21] László Toka, Gergely Dobreff, Balázs Fodor, and Balázs Sonkoly. "Machine Learning-Based Scaling Management for Kubernetes Edge Clusters". In: *IEEE Transactions on Network and Service Management* 18.1 (2021-03). Conference Name: IEEE Transactions on Network and Service Management, pp. 958–972. ISSN: 1932-4537. DOI: 10.1109/TNSM.2021.3052837.

[TG20] Salman Taherizadeh and Marko Grobelnik. “Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications”. en. In: *Advances in Engineering Software* 140 (2020-02), p. 102734. ISSN: 09659978. DOI: 10.1016/j.advengsoft.2019.102734.

[TH+21] Stefan Throner, Heiko Hutter, Niklas Sänger, Michael Schneider, Simon Hanselmann, Patrick Petrovic, and Sebastian Abeck. “An Advanced DevOps Environment for Microservice-based Applications”. en. In: *2021 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. Oxford, United Kingdom: IEEE, 2021-08, pp. 134–143. ISBN: 978-1-66543-477-5. DOI: 10.1109/SOSE52839.2021.00020.

[Ti00] Walter F Tichy. “Hints for Reviewing Empirical Work in Software Engineering”. en. In: *Empirical Software Engineering* 5.4 (2000), pp. 309–312. DOI: https://doi.org/10.1023/A:1009844119158.

[TLF-OCI] The Linux Foundation. *Open Container Initiative - Open Container Initiative*. URL: https://opencontainers.org/ (visited on 2024-07-01).

[TU+21] Songpon Teerakanok, Tetsutaro Uehara, and Atsuo Inomata. “Migrating to Zero Trust Architecture: Reviews and Challenges”. en. In: *Security and Communication Networks* 2021 (2021-05). Ed. by Qi Li, pp. 1–10. ISSN: 1939-0122, 1939-0114. DOI: 10.1155/2021/9947347.

[Vi07] Hans van Vliet. *Software Engineering: Principles and Practice*. 3rd ed. Chichester, England ; Hoboken, NJ: John Wiley & Sons, 2008. ISBN: 978-0-470-03146-9.

[VJ22] K. Vijayalakshmi and V. Jayalakshmi. “A Study on Current Research and Challenges in Attribute-based Access Control Model”. en. In: *Intelligent Data Communication Technologies and Internet of Things*. Ed. by D. Jude Hemanth, Danilo Pelusi, and Chandrasekar Vuppalapati. Vol. 101. Singapore: Springer Nature Singapore, 2022, pp. 17–31. ISBN: 9789811676093 9789811676109. DOI: 10.1007/978-981-16-7610-9_2.

[VK+23] Venčkauskas, Algimantas and Kukta, Donatas and Grigaliūnas, Šarūnas and Brūzgienė, Rasa. “Enhancing Microservices Security with Token-Based Access Control Method”. In: *Sensors* 23.6 (2023). ISSN: 1424-8220. DOI: 10.3390/s23063363.

[VS+19] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. “Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes”. en. In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. Sofia, Bulgaria: IEEE, 2019-07, pp. 176–185. ISBN: 978-1-72813-927-2. DOI: 10.1109/QRS.2019.00034.

[WA08] Luay A. Wahsheh and Jim Alves-Foss. “Security Policy Development: Towards a Life-Cycle and Logic-Based Verification Model”. en. In: *American Journal of Applied Sciences* 5.9 (2008-09), pp. 1117–1126. ISSN: 15469239. DOI: 10.3844/ajassp.2008.1117.1126.

[Wa24] Melissa A. Russell, Eric Berkowitz, Michelle Phon, and Jennie Zhu-Mai. *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), Version 4.0*. en. Ed. by Hironori Washizaki. URL: www.swebok.org.

[Wi12] Adam Wiggins. *The Twelve-Factor App*. 2012. URL: http://12factor.net (visited on 2020-09-08).

[WL+21] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. “Design, monitoring, and testing of microservices systems: The practitioners’ perspective”. en. In: *Journal of Systems and Software* 182 (2021-12), p. 111061. ISSN: 01641212. DOI: 10.1016/j.jss.2021.111061.

[WM17] Kevin Walsh and John Manferdelli. “Mechanisms for Mutual Attested Microservice Communication”. en. In: *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*. Austin Texas USA: ACM, 2017-12, pp. 59–64. ISBN: 978-1-4503-5195-9. DOI: 10.1145/3147234.3148102.

[WR+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. en. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-29043-5 978-3-642-29044-2. DOI: 10.1007/978-3-642-29044-2.

[WW+04] Lingyu Wang, Duminda Wijesekera, and Sushil Jajodia. “A logic-based framework for attribute based access control”. en. In: *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*. Washington DC USA: ACM, 2004-10, pp. 45–55. ISBN: 978-1-58113-971-6. DOI: 10.1145/1029133.1029140.

[XP+12] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. “Automated extraction of security policies from natural-language software documents”. en. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. Cary North Carolina: ACM, 2012-11, pp. 1–11. ISBN: 978-1-4503-1614-9. DOI: 10.1145/2393596.2393608.

[XZ+23] Shaowen Xu, Qihang Zhou, Heqing Huang, Xiaoqi Jia, Haichao Du, Yang Chen, and Yamin Xie. “Log2Policy: An Approach to Generate Fine-Grained Access Control Rules for Microservices from Scratch”. en. In: *Annual Computer Security Applications Conference*. Austin TX USA: ACM, 2023-12, pp. 229–240. ISBN: 9798400708862. DOI: 10.1145/3627106.3627137.

[YB18] Tetiana Yarygina and Anya Helene Bagge. “Overcoming Security Challenges in Microservice Architectures”. In: *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. 2018-03, pp. 11–20. DOI: 10.1109/SOSE.2018.00011.

[YO+23] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. *Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT*. en. arXiv:2304.10778 [cs]. 2023-10. URL: http://arxiv.org/abs/2304.10778 (visited on 2024-11-07).

[YT05] E. Yuan and J. Tong. “Attributed based access control (ABAC) for Web services”. In: *IEEE International Conference on Web Services (ICWS'05)*. 2005-07, p. 569. DOI: 10.1109/ICWS.2005.25.

[Zi17] Olaf Zimmermann. “Microservices tenets: Agile approach to service development and deployment”. en. In: *Computer Science - Research and Development* 32.3-4 (2017-07), pp. 301–310. ISSN: 1865-2034, 1865-2042. DOI: 10.1007/s00450-016-0337-0.

[ZO+23] Ahmed Zerouali, Ruben Opdebeeck, and Coen De Roover. “Helm Charts for Kubernetes Applications: Evolution, Outdatedness and Security Risks”. In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. ISSN: 2574-3864. 2023-05, pp. 523–533. DOI: 10.1109/MSR59073.2023.00078.

[ZP+18] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. “Benchmarking microservice systems for software engineering research”. en. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. Gothenburg Sweden: ACM, 2018-05, pp. 323–324. ISBN: 978-1-4503-5663-3. DOI: 10.1145/3183440.3194991.

[ZZ+24] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. *A Survey of Large Language Models*. en. arXiv:2303.18223 [cs]. 2024-10. URL: http://arxiv.org/abs/2303.18223 (visited on 2024-11-07).