Karlsruhe Institute of Technology

Software Design and Quality

Guided Exploration and Visualization of Trace Links in Visual Studio Code

Bachelor's Thesis of

Julian Robin Winter

At the KIT Department of Informatics
KASTEL – Institute of Information Security and Dependability

First examiner:	Prof. Dr. Ralf H. Reussner
Second examiner:	Prof. Dr.-Ing. Anne Koziolek
First advisor:	Dr. Kevin Feichtinger
Second advisor:	Dominik Fuchß, M.Sc.

July 21st 2025 – November 21st 2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Guided Exploration and Visualization of Trace Links in Visual Studio Code (Bachelor's Thesis)

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, November 21st 2025

. . . . . .

(Julian Robin Winter)

Abstract

Modern software systems rely on numerous interconnected artifacts: requirements, documentation, models, and code. Traceability links between these artifacts support comprehension, maintenance, and onboarding of new personnel. While automated Trace Link Recovery (TLR) approaches such as ARDoCo and LiSSA can generate these links, developers rarely use them effectively because they are not integrated into their daily development environment and workflows. This thesis introduces *trace-viz*, a Visual Studio Code extension that integrates automatically generated or imported trace links directly into the IDE. The extension supports multiple trace visualization modes like sentence-to-file. It enables interactive exploration of trace links through inline markers, context tooltips and quick navigation. *Trace-viz* was developed to have a modular and expandable architecture. A think-aloud evaluation using software comprehension tasks showed that *trace-viz* improves developers efficiency and accuracy when working with trace links. Participants reported lower navigation effort, better understanding of artifact relationships, increased confidence identifying relevant files and were able to save time with their task. Compared to using trace links in raw CSV format, the extension led to more consistent task completion and reduced cognitive load. The results demonstrate that embedding trace-link visualization directly into the IDE provides meaningful support for software tasks, comprehension and maintenance. *Trace-viz* successfully bridges the gap between automated TLR approaches and real-world developer workflows. Future work should include expanding visualization methods and conducting broader evaluations across diverse project types.

Zusammenfassung

Moderne Softwaresysteme basieren auf zahlreichen miteinander verknüpften Artefakten: Anforderungen, Dokumentation, Modelle und Code. Traceability Links zwischen diesen Artefakten unterstützen das Verständnis, die Wartung und das Onboarding. Obwohl automatisierte Trace Link Recovery (TLR) Ansätze wie ARDoCo und LiSSA solche Links erzeugen können, nutzen Entwickler sie selten effektiv, da sie nicht in ihre tägliche Entwicklungsumgebung und Arbeitsabläufe integriert sind. Diese Arbeit stellt *trace-viz* vor, eine Visual-Studio-Code-Erweiterung, die automatisch erzeugte oder importierte Trace Links direkt in die IDE integriert. Die Erweiterung unterstützt verschiedene Trace-Visualisierungsmodi, wie etwa „Satz-zu-Datei“. Sie ermöglicht eine interaktive Erkundung der Trace Links durch Inline-Markierungen, kontextbezogene Tooltips und schnelle Navigation. *Trace-viz* wurde mit einer modularen und erweiterbaren Architektur entwickelt. Eine Think-Aloud-Evaluation anhand von Softwareverständnisaufgaben zeigte, dass *trace-viz* die Effizienz und Genauigkeit von Entwicklern beim Arbeiten mit Trace Links verbessert. Teilnehmende berichteten von geringerem Navigationsaufwand, einem besseren Verständnis der Artefaktbeziehungen und höherer Sicherheit bei der Identifikation relevanter Dateien. Im Vergleich zur Nutzung von Trace Links im reinen CSV-Format führte die Erweiterung zu konsistenteren Aufgabenergebnissen und einer reduzierten kognitiven Belastung. Die Ergebnisse zeigen, dass die direkte Einbettung von Trace-Link-Visualisierung in die IDE eine bedeutende Unterstützung für Softwareaufgaben, Verständnis und Wartung bietet. *Trace-viz* schließt erfolgreich die Lücke zwischen automatisierten TLR-Ansätzen und realen Entwicklerworkflows. Zukünftige Arbeiten sollten die Erweiterung der Visualisierungsmethoden und umfassendere Evaluierungen über verschiedene Projekttypen umfassen.

Contents

List of Figures

List of Tables

1. Introduction

Modern software systems generate a wide range of diverse development artifacts, including natural-language documentation, architectural models, and source code. Maintaining traceability links between these artifacts is essential for tasks such as program comprehension, change impact analysis, and quality assurance. While research frameworks such as ARDoCo [21] and LiSSA [10] provide advanced methods for automatically recovering trace links, their outputs are typically consumed outside of the primary development environment. As a result, developers must frequently switch between tools, formats, and representations, which interrupts their workflow and limits the practical usefulness of traceability information. This thesis is situated within this gap: the need to make automatically generated trace links accessible, interactive, and directly integrated into the everyday work of software developers.

1.1. Motivation

Despite the availability of powerful trace link recovery approaches, the exploration of traceability information remains challenging in real-world development settings. Existing solutions often rely on static files such as CSV exports or external dashboards and tools, which provide little guidance to navigate relationships between documentation, models, and code. Developers face difficulties interpreting large trace link sets, understanding how artifacts relate, and locating relevant elements efficiently. Consequently, trace links—although valuable—are frequently underutilized. To address this problem, this thesis seeks to improve developer effectiveness and efficiency by embedding interactive, fine-grained trace link exploration directly into the Visual Studio Code environment. The overarching goal therefore is to evaluate to what extent an integrated Visual Studio Code extension for guided exploration and visualization of trace links improve developers effectiveness and efficiency in understanding and navigating relationships between documentation, models, and code, compared to having access to the same trace links in a non-visualized, unstructured format. From this goal, the following research questions are derived:

RQ1: *Does an integrated Visual Studio Code extension for guided exploration and visualization of trace links improve developers effectiveness compared to having access to the same trace links in a non-visualized format?*

RQ2: *Does visualization of trace links enhance developers efficiency in understanding and navigating relationships between documentation, models, and code?*

1.2. Contribution

To investigate these questions, this thesis presents *trace-viz*, a Visual Studio Code extension that integrates imported or automatically generated trace links from ARDoCo and LiSSA and visualizes them directly within the editor. The extension provides several visualization modes, like sentence-to-file that enable developers to navigate linked artifacts through gutter markers, quick-access menus, and contextual tooltips.

1.3. Evaluation Method

Beyond the implementation of a modular and extensible architecture, this thesis contributes a systematic empirical evaluation conducted through a think-aloud user study. The evaluation examines whether visualization improves navigation efficiency, task success, and program comprehension when compared to working with raw CSV trace links. Qualitative and quantitative results provide insights into user behavior, perceived usefulness, and the practical value of integrating traceability into an IDE.

1.4. Structure of the Thesis

After the introduction Chapter 2 introduces the theoretical foundations of software traceability, discussing trace link types, recovery approaches, and supporting frameworks such as ARDoCo and LiSSA, as well as related work on trace link visualization and existing IDE-based approaches. Chapter 3 describes the architecture, functionality, and implementation of the *trace-viz* extension. Chapter 4 outlines the evaluation design, including study methodology, research questions, and data collection procedures. Chapter 5 presents and discusses the results of the user study, combining quantitative metrics with qualitative insights from think-aloud transcripts. Finally, Chapter 6 summarizes the key findings, addresses limitations, and outlines opportunities for future work.

2. Literature Review on Traceability and Visualization

This chapter presents the scientific and methodological foundations of this bachelor's thesis. It summarizes the relevant theoretical background, defines key concepts essential for understanding the work, and discusses prior research in the field. In addition, it provides an overview of the tools, frameworks, and technologies used during the development and evaluation of the proposed solution, as well as related work on trace link visualization and existing IDE-based approaches.

2.1. Traceability and Trace Links

Software traceability refers to the ability to establish and maintain connections between the various artifacts produced throughout the software development lifecycle. These artifacts include requirements, architectural models, source code, test cases, and different forms of documentation. Trace links represent explicit relationships between such artifacts and are essential for tasks including change impact analysis, validation and verification, and regulatory compliance [8].

2.2. Types of Trace Links

Trace links can be classified according to the kinds of artifacts they relate. In this thesis, the primary focus lies on documentation-to-model, model-to-code, and documentation-to-code trace links. Beyond these, additional categories of trace links exist, which can be grouped according to the artifact types they connect:

A first group comprises links originating from requirements. These include requirements-to-design links that relate requirements to architectural or design elements, requirements-to-code links that connect them to implementing source code units, and requirements-to-test links that associate them with verifying test cases. Requirements-to-requirements links capture refinement or hierarchical relationships between high-level and more detailed requirements.

A second group consists of links between design artifacts and implementation. Design-to-code links describe relationships between architectural or design models and the code units

that realize them, whereas code-to-test links relate implementation code to the test cases that exercise it.

A third group includes links involving natural-language documentation. Documentation-to-code links connect informal textual descriptions with the corresponding code elements, and documentation-to-model links relate documentation statements to architectural or design models.

2.3. Trace Link Recovery Approaches

Trace Link Recovery (TLR): Trace links may be established manually, semi-automatically, or fully automatically. Manual traceability is often accurate but time-consuming and prone to human error [13, 2]. Information retrieval (IR)-based techniques suggest links by measuring textual similarity between artifacts [1]. Natural language processing methods, such as those employed in ARDoCo, analyse linguistic structures to identify semantically related elements across documents, models, and code [14]. More recently, machine learning and deep learning approaches have been applied to capture complex semantic relationships, although they typically require substantial training data and careful configuration.

The increased adoption of model-based and NLP-supported approaches aims to reduce manual effort while maintaining or improving the quality of recovered links [6]. In the following, the effectiveness of the individual methods is compared using the F1-score, a standard metric that balances precision and recall to provide a single measure of overall accuracy in trace link recovery.

2.4. ARDoCo: Automating Requirements and Documentation Comprehension

ARDoCo (Automating Requirements and Documentation Comprehension)[21] is an open-source research framework that supports traceability link recovery and consistency checking across heterogeneous software artifacts. Its primary goal is to bridge the semantic gap between informal natural-language documentation and more formal artifacts such as software architecture models or source code. To achieve this, ARDoCo integrates classical natural language processing techniques, heuristic matching strategies, and domain-specific modeling concepts to identify, propose, and evaluate trace links. ARDoCo is actively developed by researchers of the *Modelling for Continuous Software Engineering (MCSE)*[1] group at the *KASTEL – Institute of Information Security and Dependability*[2] at the Karlsruhe Institute of Technology (KIT). The long-term research vision behind ARDoCo is to maintain consistency between the various artifacts created during software development and evolution

[1] https://mcse.kastel.kit.edu
[2] https://www.kastel.kit.edu

(even whiteboard discussions). Thereby preserving architectural knowledge and preventing information loss as systems evolve.

A central component of ARDoCo is its unified REST API[3], which exposes all traceability link recovery pipelines implemented within the framework. The API offers access to several specialised TLR methods that address different combinations of software artifacts. Including documentation-to-model, model-to-code, and documentation-to-code link recovery. These pipelines correspond directly to the core approaches discussed in the following sections. Through a uniform interface, each method can be started, monitored, and retrieved, enabling seamless integration of ARDoCo into external tools such as IDE plugins, automated analysis workflows, and research prototypes. The API supports all major variants of the framework, including *SWATTR* for linking architecture documentation to architecture models (SAD-SAM). *ArCoTL* for connecting models to source code (SAM-Code). *ArDoCode* for directly linking documentation to code (SAD-Code). Furthermore *TransArC* for combining documentation–model and model–code links in a transitive workflow (SAD-SAM-Code).

2.4.1. ArDoCode

ArDoCode provides traceability link recovery between software architecture documentation and source code (SAD-Code). Instead of using a formal architecture model as an intermediate representation, ArDoCode treats the code itself as the target model and aligns concepts extracted from the documentation with identifiers in the codebase. It applies the same heuristic principles as SWATTR but without requiring a structured Software Architecture Model (SAM). Because it omits the formal modeling step, ArDoCode is straightforward to apply but considerably less precise. Empirical evaluations report a weighted F1-score of roughly 0.62, which is substantially lower than model-based approaches such as TransArC. As a result, ArDoCode mainly serves as a baseline, illustrating that the absence of structured architectural models leads to a noticeable drop in TLR performance. [16]

2.4.2. TransArC

TransArC performs traceability link recovery between software architecture documentation, architecture models, and code (SAD-SAM-Code). It follows a transitive strategy that first links documentation to a Software Architecture Model (SAM) using SWATTR and subsequently links the SAM to source code using ArCoTL. By composing these two link sets, TransArC establishes documentation-to-code links through the sequence *documentation* $\rightarrow$ *model* $\rightarrow$ *code*. This two-step process effectively reduces the semantic distance between textual descriptions and low-level code. Experimental studies on multiple systems showed that TransArC achieves an average F1-score of around 0.82, significantly outperforming single-step approaches such as ArDoCode. [9, 16]

[3] https://rest.ardoco.de/swagger-ui/index.html

2.4.3. SWATTR

SWATTR (SoftWare Architecture Text Trace link Recovery) links textual software architecture documentation with corresponding architecture models (SAD-SAM). It follows an extensible agent-based pipeline in which textual elements and model components are extracted, processed, and matched using natural language processing and heuristic reasoning. SWATTR analyses sentences from the documentation to identify mentions of architecture-relevant concepts and aligns them with components of the Software Architecture Model. Evaluations across several case studies demonstrated strong performance, with a weighted average F1-score of about 0.72, surpassing simpler baselines. [17, 15, 16]

2.4.4. ArCoTL

ArCoTL (Architecture–Code Trace Links) recovers links between architecture models and the source code (SAM-Code). It transforms both the architecture model and the implementation into simplified internal representations and applies naming similarities, structural relationships, and dependency-based heuristics to identify corresponding elements. By combining multiple independent and relationship-aware strategies, ArCoTL provides highly accurate trace links. Empirical evaluations reported an average F1-score of approximately 0.98, indicating excellent effectiveness for model-to-code alignment. [16]

2.4.5. LiSSA: Linking Software System Artifacts

LiSSA (Linking Software System Artifacts) is an LLM- and retrieval-augmented approach to traceability link recovery that aims to be generic across different artifact types. Its central idea is to combine information retrieval with the reasoning capabilities of Large Language Models (LLMs) to identify meaningful connections between heterogeneous software artifacts. For any given source element, such as a requirement or a sentence in architecture documentation, LiSSA first retrieves a reduced set of potentially relevant target artifacts by means of embedding-based similarity search. This retrieval step narrows the search space and provides contextual evidence. The retrieved candidates together with the source artifact are then provided to an LLM, which evaluates the semantic relationship and decides whether a trace link should be established. In this way, LiSSA applies a consistent retrieval-augmented generation process across different artifact pairs.

LiSSA has been evaluated on several traceability tasks, including requirements-to-code, documentation-to-code, and architecture documentation-to-model links. The same retrieval-based workflow is used for each of these scenarios, demonstrating that LiSSA can act as a general-purpose traceability framework rather than a technique specialized for a single artifact combination. Experimental results show that LiSSA achieves substantially higher accuracy than traditional approaches, particularly in code-centric scenarios. In tasks such as requirements-to-code tracing, LiSSA outperformed state-of-the-art baselines by a considerable margin, highlighting the potential of LLM-based reasoning combined with

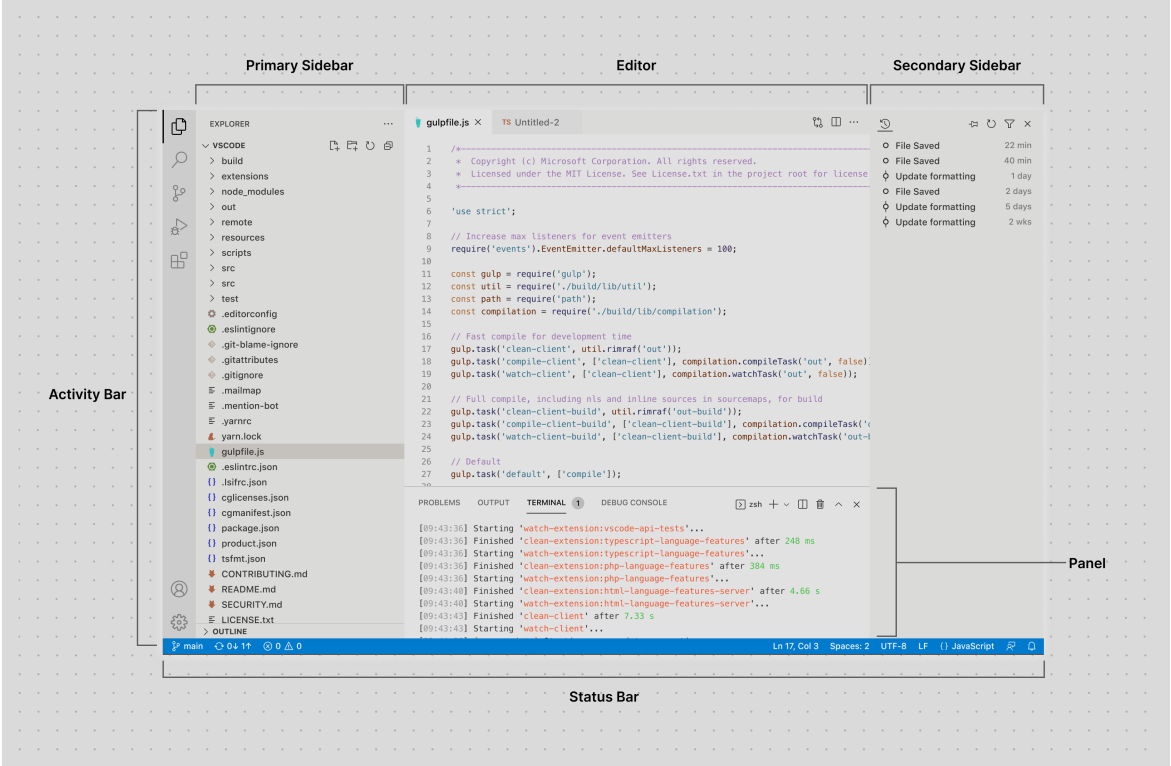


Figure 2.1.: Overview of the VSCode interface [20]

retrieval to improve the flexibility and effectiveness of automated traceability link recovery. [8, 11, 13]

2.5. Visual Studio Code and Extension Development

Visual Studio Code (VS Code) is a widely used, lightweight, and extensible source-code editor. Through its Extension API[4] developers can create custom plugins that enhance the IDE's capabilities by adding views, UI elements, and programmatic interactions with the workspace and files.

Relevant features for this thesis include the extension contributions to the Activity Bar, the Status Bar, views (especially tree views) and editor-adjacent UI elements such as gutter decorations, CodeLens and hover tool-tips. Below are the main extension entry points and UX-guideline considerations:

Activity Bar

The Activity Bar is the vertical toolbar typically positioned on the left side of the VS Code workbench as seen in Figure 2.1. Extensions may contribute new view-containers as Activity Bar items. According to the UX guidelines, extensions should use an icon in the same style

[4] https://code.visualstudio.com/api

as native Activity Bar items and supply a clear and concise name for the view container. Duplicate icons or mis-styled icons are discouraged.

Status Bar

The Status Bar runs along the bottom of the workbench and is designed to display contextual information about the workspace or the active file. The UX guidelines specify that items which relate to the entire workspace (e.g., global status) should appear on the left, while items scoped to the active file or context should appear on the right. It is recommended to limit the number of contributed items, to use short text labels, and to avoid custom colours unless absolutely necessary.

Views and Tree Views

Extensions may contribute custom views that appear in the Sidebar or Panel. A view can be of type Tree View (hierarchical data), Welcome View, or Webview View. The UX guidelines recommend limiting the number of views to avoid clutter, ensuring buttons and actions are used only for primary operations, and aligning icons with the native icon library where possible. Tree views in particular allow hierarchical data, are backed by a TreeDataProvider, and must implement the appropriate API registration in the extension manifest.

Editor-Adjacent Elements: Gutter, CodeLens and Hover

Although not separate view containers, several UI affordances enable rich in-editor interactions. Gutter decorations allow the extension to place icons or markers beside line numbers; CodeLens provides inline actionable links above code lines; and hover tool-tips enable contextual information when a user hovers over identifiers or code elements. These elements allow the extension to embed traceability link information, quick-actions, or visual cues within the main editor space.

UX Foundations and Restrictions

The UX guidelines emphasise that extensions should "fit seamlessly into the VS Code user interface" and follow preferred workflows. By aligning the extension's UI contributions with these guidelines, the implementation of this thesis's VS Code plugin can provide a consistent and intuitive user experience while integrating trace-link visualisation, interactive trace navigation and user feedback mechanisms.

2.6. Trace-Link Visualization

Research in the area of software traceability has highlighted the importance of providing not just trace links but also effective ways to explore and visualize them. Various approaches have been proposed to assist developers in navigating complex trace relationships through guided exploration and interactive visualization.

Traditional visualization methods for software traceability primarily rely on matrix and cross-reference techniques. Traceability matrices use a two-dimensional grid to map relationships

between artifacts, offering a simple and intuitive way to inspect trace links when the number of artifacts is small [24]. However, matrix-based approaches do not capture hierarchical structures and quickly become cluttered as the number of artifacts grows, limiting their scalability.
Chen et al. (2012) [5] introduced an early approach combining treemap and tree-based visualizations to enable scalable browsing of trace links between code and documentation. Their work demonstrated that combining global overviews with focused details reduced visual clutter and improved developer comprehension during maintenance tasks.
Aung et al. (2019) presented Hierarchical Trace Maps, visualizing trace links across multiple types of software artifacts, including requirements, code, and tests. Their method enabled developers to interactively filter, expand, and traverse trace links in layered structures, supporting program comprehension and change impact analysis [3].

Kugele and Antkowiak (2016) proposed a dual-level visualization approach designed to make large and complex trace link networks more comprehensible [18]. Their method combines a global overview with a local, detail-oriented graph representation. The global view is based on the "Impact City" metaphor, where artifacts are visually organized into buildings and districts. Building height, size, and color encode properties such as artifact importance and link density, enabling engineers to grasp system structure and potential change impact at a glance. Complementing this, the local view provides a layered graph-based representation, allowing users to zoom into individual artifacts and inspect their direct trace links in detail. This two-tier visualization strategy aims to support engineers in understanding change propagation paths, even when dealing with thousands of artifacts. The authors formulated two key research questions for future validation: (i) How useful is the visualization concept for engineers in real industrial projects? (ii) Does the computed impact analysis accurately reflect the engineer's own assessment? For this thesis, the Impact City concept serves as inspiration for providing a high-level overview of traceability within the Visual Studio Code extension. Similar to the city metaphor, clusters of related files, like directories can presented as aggregated units to reduce cognitive load. The detailed graph-based view aligns with the extension's local, artifact-centered exploration, enabling developers to inspect precise trace links on demand. This "overview-first, zoom-and-filter, details-on-demand" pattern supports intuitive navigation of traceability information and prevents developers from being overwhelmed by large link networks.

Despite these advances, few tools have embedded guided trace link exploration directly into integrated development environments (IDEs). Industrial solutions, such as Reqtify [7], provide trace link visualization but are often external to IDEs or focused on compliance contexts. Extensions like CodeGraphy [25] or AtomicViz [19] have brought interactive graph-based navigation into IDEs but focus on code-to-code relationships rather than traceability between documentation, design, and code artifacts.

In summary, prior research has shown the benefits of multi-level visualizations (overview+detail), interactive filtering, and even explanation-enhanced trace links. However, there remains a significant gap in embedding these features directly within developers working environments. Particularly in lightweight IDEs like Visual Studio Code. This thesis addresses this

gap by integrating multiple guided visualization strategies into a VS Code extension that supports extensible trace link sources.

3. Approach

The goal of this thesis is to explore whether the visualization and guided exploration of trace links within an Integrated Development Environment (IDE) can improve developer's ability to understand and work with software systems. Specifically, this thesis focuses on designing and implementing a Visual Studio Code (VS Code) extension that enables developers to view and interact with traceability information directly within their workflow. The extension received the name *trace-viz*

Traceability information plays a central role in software development and maintenance, particularly in understanding the relationship between documentation, models, and code. While frameworks such as ARDoCo [21] and LiSSA [10] provide powerful methods for generating trace links automatically, they lack direct integration into the developer's primary work environment. This thesis bridges that gap by providing a modular, extensible VS Code extension that not only supports multiple sources of trace link generation but also offers different levels of visualization granularity.

3.1. Usage Scenario and System Workflow

From the user's perspective, the extension is designed to fit seamlessly into the development process. Once installed, the extension can be activated within any project. Depending on the project not all traceability approaches can be used. ARDoCo only works with Java files. LiSSA uses Large Language Models (LLMs) and Retrieval-Augmented Generation (RAG) and can thus be used in every project. The user then provides input artifacts depending on the type of traceability they wish to generate. These inputs may include natural language documentation, such as requirements or architecture descriptions, and architecture models, for example UML diagrams.

The extension *trace-viz* can be found in the activity bar[1], indicated by a link icon, as shown in Figure 3.1. The *trace-viz* extension UI is split into two views[2]. The first is the traceability approach. Here, different traceability approaches are available for the user to generate trace links. Clicking on an approach either opens a configuration view (as shown in Figure 3.2 for the ARDoCo config) or opens a tree view. This enables multiple steps that must be completed before trace-link generation.

[1] https://code.visualstudio.com/api/ux-guidelines/activity-bar
[2] https://code.visualstudio.com/api/ux-guidelines/views

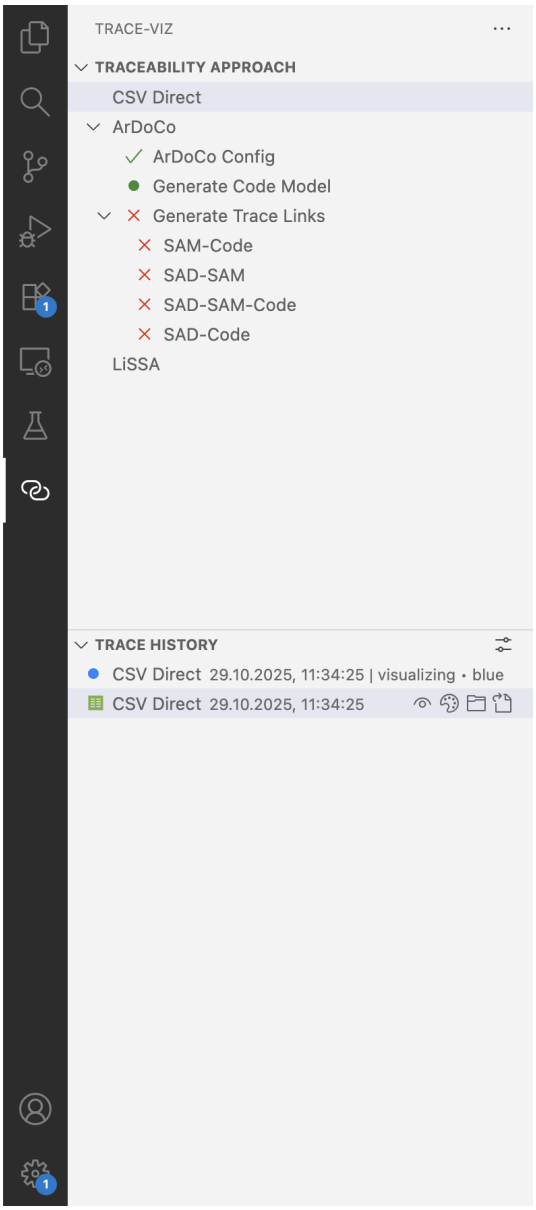


Figure 3.1.: trace-viz in activity bar and split UI of traceability approach and trace history

To provide feedback about the current status, the extension uses a set of symbols, as shown in Figure 3.3. A green checkmark symbolizes that the step is completed. A green dot indicates that preconditions are met but the step is not yet finished. A red X indicates that the step is not complete or that its preconditions have not yet been met.

For example, ARDoCo requires a completed configuration and a generated code model before different types of trace links can be produced, depending on the available or specified artifacts.

The bottom view is the Trace History View. Here the generated trace links appear once generated. Clicking on the trace history line the corresponding artifact (documentation)

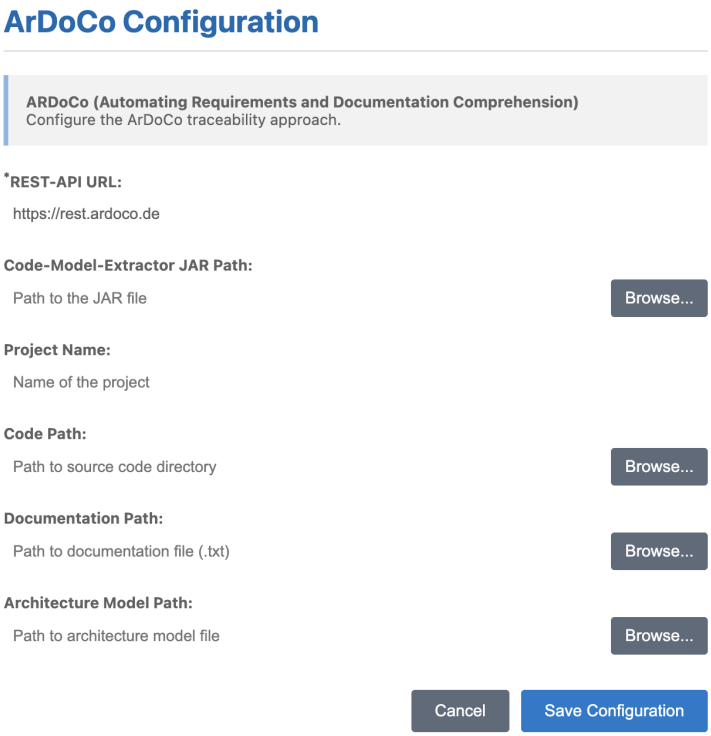


Figure 3.2.: ARDoCo Config

Figure 3.3.: Status indicators

opens and by clicking the eye symbol activates the visualization. As seen in Figure 3.5 all lines with a trace-link get marked by a dot in the gutter. Up to 2 trace links history items can be visualized at once. The dots can be switched between blue, red, yellow and green to differentiate and compare the different trace links.

Based on the selected configuration, trace links can either be imported directly from a CSV file or generated using trace link recovery approaches such as ARDoCo or LiSSA. In the case of direct CSV import, the user provides pre-existing trace links, for example originating from prior research or manual annotation. Through ARDoCo integration via a REST API, the extension communicates with the ARDoCo backend to automatically derive trace links. Alternatively, LiSSA integration via a JAR interface enables the execution of a local JAR file through the command line to generate trace links using retrieval-augmented generation.

Internally, the extension analyses the project's code and produces an intermediate representation in the form of an ACM file, which ensures that external tools such as ARDoCo can correctly interpret and process the provided artifacts.

This design illustrates the extensibility of the extension, as additional trace link sources can be incorporated with minimal effort. The architecture accommodates file-based, API-based, and CLI-based integration, thereby enabling the system to evolve alongside future tooling developments.

3.2. Architecture Overview

Trace-viz is a VS Code extension that generates, manages, and visualizes trace links between documentation, software artifacts, and source code. The extension follows a layered, modular architecture that separates user interface concerns from business logic and data management. This design promotes maintainability and enables straightforward extension with new traceability approaches.

The architecture consists of four primary layers: the *Presentation Layer*, which handles user interaction through VS Code's UI components; the *Application Layer*, which orchestrates user commands and coordinates between components; the *Domain Layer*, which contains business logic and state management; and the *Integration Layer*, which interfaces with external tools and handles data persistence. These layers communicate through well-defined interfaces, ensuring loose coupling and high cohesion.

3.2.1. Component Structure

The extension is organized into distinct architectural layers, each serving a specific purpose in the overall system as seen in Figure 3.4:

Presentation Layer: This layer manages all user-facing components. The Tree Views component provides two main views: the Traceability Approach view, which displays available trace link generation methods and their status, and the Trace History view, which shows past trace link generation runs. The Visualization Engine component renders trace links directly in the editor through gutter decorations, status bar indicators, and interactive hover tooltips. This layer reacts to user interactions and displays state changes from the underlying layers.

Application Layer: The Commands component acts as the central orchestrator, translating user actions from the presentation layer into operations on the domain layer. Commands handle user requests such as opening files, toggling visualizations, generating trace links, and managing configurations. This layer maintains no state itself but coordinates between the presentation and domain layers.

Domain Layer: This layer contains the core business logic. The Services component manages application state, including trace history persistence, visualization state, and

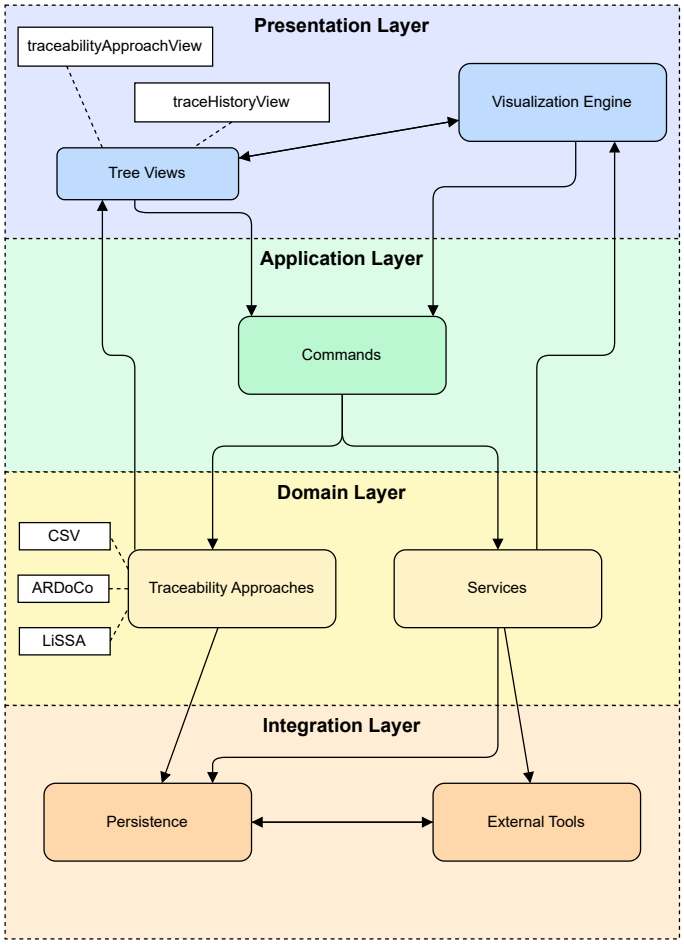


Figure 3.4.: High-level architecture showing the four main layers and their relationships

configuration management. The Traceability Approaches component encapsulates different methods for generating trace links, such as ARDoCo integration and CSV import. Each approach is self-contained, allowing new traceability methods to be added without modifying existing code.

Integration Layer: This layer handles external interactions and data persistence. The Persistence component manages workspace-local storage, reading and writing configuration files, trace history, and generated artifacts. The External Tools component interfaces with external systems, such as the ARDoCo REST API and the Code Model Extractor JAR, to generate trace links and code models.

`Legend:` The diagram uses rounded rectangles to represent the system's modules and components, arranged within colored horizontal bands that denote the four architectural layers (Presentation, Application, Domain, and Integration). A dashed outer border marks the overall system boundary. Solid arrows indicate data or control flow between components,

while curved connector lines serve purely as visual routing aids without additional semantic meaning. White boxes connected by dotted lines illustrate instantiated elements of the respective modules and components.

3.2.2. Extensibility

The architecture is designed to support extension in several key areas:

New Traceability Approaches: The modular structure of the Traceability Approaches component allows new trace link generation methods to be added by creating a new module under the traceability approach directory. Each module implements a standard interface for configuration, generation, and result handling, ensuring consistent integration with the rest of the system.

UI Extensions: The Tree View providers use VS Code's TreeDataProvider interface, which allows new UI elements to be added by extending existing providers or creating new ones. The reactive update mechanism ensures that new UI components automatically stay synchronized with underlying data changes.

Service Extensions: The service layer uses a singleton pattern with a centralized access point, making it straightforward to add new services for additional functionality. Services can be extended or replaced without affecting other components, as long as they maintain the expected interface.

Storage Backends: The persistence layer abstracts file system operations, allowing the storage mechanism to be extended or replaced. While currently using workspace-local JSON files, the architecture supports future extensions to different storage backends, such as databases or cloud storage, without requiring changes to the domain layer.

Event-Driven Architecture: Components communicate through VS Code's event system and custom event emitters. This reactive approach allows new components to subscribe to relevant events without modifying existing code, promoting loose coupling and enabling incremental feature additions.

3.3. Visualization Modes

The generated or imported trace links are stored in a structured `.csv` format and visualized within the IDE using different modes. The file-to-file view shows relations between entire documents and code files, for example from a specification document to a Java class file. The sentence-to-file view offers fine-grained inspection of how individual sentences in documentation relate to code elements. The UML-to-file view displays links between UML elements, such as class diagram entities, and the corresponding implementation in code.

For the sentence-to-file view, multiple visualization mechanisms are available to display trace links. Lines that contain one or more trace links are marked with a colored indicator in

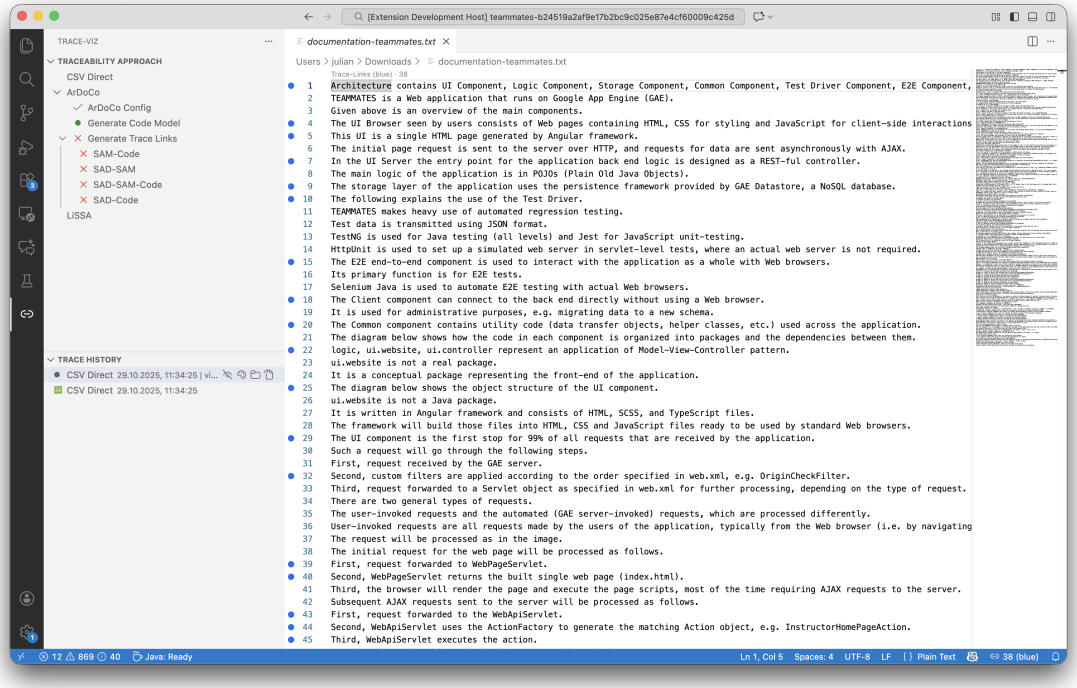


Figure 3.5.: Colored dots besides line numbers

the gutter. The associated trace links can be accessed by hovering over the marked line and selecting the corresponding entry in the Quick Pick menu (see Figure 3.6). Alternatively, the same functionality can be triggered by selecting the CodeLens annotation that appears above the line or by using the dedicated button in the status bar located in the lower-right corner of the editor interface (see bottom of Figure 3.5).

3.4. Implementation

The extension is implemented in TypeScript using the Visual Studio Code Extension API. It follows a modular architecture that ensures clear separation of concerns and future extensibility.

At the core of the extension is the Input Handling Module, which is responsible for managing different types of inputs. Users can either load existing trace links through CSV files, generate new trace links via the ARDoCo REST API, or run the LiSSA tool locally through a JAR file. Depending on the selected mode, the input module processes various project artifacts, such as documentation, UML diagrams, or source code.

The User Interface is designed to provide multiple ways to explore trace links inside Visual Studio Code. Trace links are presented in a status view that lists all traceable connections and allows navigation between linked artifacts through the quick pick menu. Additionally, inline annotations are shown directly in the code editor.

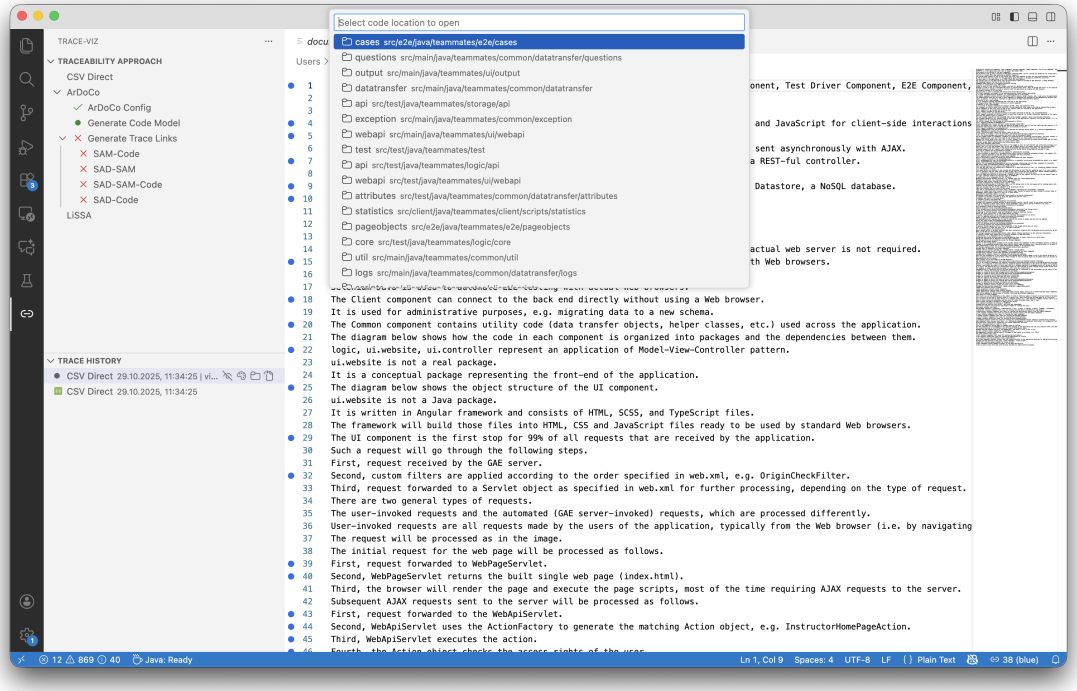


Figure 3.6.: Quick picks menu

Finally, the Configuration and Settings Layer provides flexibility for users to adapt the extension to their needs. Users can configure the trace link generation mode, define input paths, and select preferred visualization options. This modular and user-centric architecture ensures that the extension remains both flexible and scalable, allowing additional trace link sources or visualization methods to be integrated in future work.

3.5. Design Rationale

Several design decisions were made to ensure that the extension aligns with the underlying research objectives. Visual Studio Code was chosen as the target IDE because of its widespread adoption, extensive extension ecosystem, and well-documented API, all of which facilitate both implementation and empirical evaluation. The extension is implemented in TypeScript, which provides static typing and advanced tooling support, resulting in improved code quality, maintainability, and robustness compared to plain JavaScript. For interoperability, CSV was selected as the exchange format for trace links, as it is simple, tool-agnostic, human-readable, and supported by both the external tools and the extension, making it an appropriate format for integrating heterogeneous traceability sources. Furthermore, the extension introduces multiple visualization modes to accommodate varying levels of analytical granularity: the file-to-file, sentence-to-file, and UML-to-file views collectively enable a continuum from high-level overviews to detailed inspection of individual trace links, thereby supporting a broad range of usage scenarios.

3.6. Assumptions and Limitations

The proposed approach operates under several assumptions. It presupposes that the software projects under consideration contain sufficiently detailed documentation, models, or related artifacts from which meaningful trace links can be generated. In addition, it assumes that the external components used for trace link extraction—namely the ARDoCo backend and the LiSSA JAR—are correctly configured, reachable, and functioning as intended whenever their respective providers are invoked within the system. At its current stage, the approach also exhibits certain limitations. Only a limited set of traceability extraction and visualization mechanisms has been implemented, which narrows the range of supported scenarios.

4. Evaluation

This chapter presents the evaluation of the developed Visual Studio Code (VS Code) extension for visualizing trace links. The evaluation aims to determine whether the extension with the visualization improves developers effectiveness and efficiency when exploring and understanding traceability relationships between documentation, models, and code.

To systematically assess the tool, the study follows the Goal–Question–Metric (GQM) framework [4] and applies a think-aloud method [23] during user testing. The findings are analyzed both qualitatively and quantitatively to identify strengths, weaknesses, and opportunities for improvement.

4.1. Goal–Question–Metric Framework

The GQM approach provides a top-down method for aligning software measurement with organizational goals by defining a clear goal, refining it into questions that characterize the object of study, and deriving objective or subjective metrics to answer these questions. The authors illustrate the hierarchical Goal→Question→Metric model using an example on improving change request processing and emphasize the importance of selecting appropriate viewpoints, formalizing processes, and iteratively refining the model. Situated within frameworks such as the Quality Improvement Paradigm and the Experience Factory, GQM has proven adaptable across diverse organizations including NASA, Hewlett-Packard, and Motorola. [23]

The GQM framework (see Table 4.1) provides a structured approach to connect high-level research goals to concrete questions and measurable outcomes. The overarching goal of this evaluation is to evaluate the usefulness and usability of the developed VS Code extension for trace link visualization in supporting software comprehension and navigation tasks.

From this goal, the following research questions are derived:

RQ1: *Does an integrated Visual Studio Code extension for guided exploration and visualization of trace links improve developers effectiveness compared to having access to the same trace links in a non-visualized format?*

RQ2: *Does visualization of trace links enhance developers efficiency in understanding and navigating relationships between documentation, models, and code?*

To address these questions, the following metrics were considered: task completion time, number of correctly identified trace elements, user-reported satisfaction and perceived

Table 4.1.: Goal–Question–Metric (GQM) Model for Evaluation

Goal	Question	Metric
Evaluate the usefulness of trace link visualization in VS Code	- Does the visualization help participants identify relevant code elements faster and more accurately? (objective) - Do users perceive the trace link visualization as helpful and worth using in practice? (subjective)	• Task completion time • Task success rate • Post-task Likert scale questions • Open feedback comments
Assess the extension's impact on program comprehension	Do users find it easier to understand the system architecture with trace links?	• Number of navigation steps • Correct identification of linked elements

usefulness, and qualitative observations from the think-aloud sessions. In the following, the study design and process are introduced. The results of the evaluation are presented and discussed in chapter 5.

4.2. Study Design and Pilot Study

The evaluation employed a think-aloud protocol in which participants verbalized their thoughts while completing predefined tasks. This method provides rich insights into users reasoning processes, navigation strategies, and the challenges they encounter when using the extension.

Because the research question focuses on effectiveness in realistic development contexts, participants completed two typical software comprehension tasks using an actual software project. Their actions and verbalizations were recorded for later analysis.

Several variations in study design were considered, including different task sequences, alternative uses of the visualization, and multiple software projects. However, given the limited scope of this thesis, a single consistent study design was chosen. All participants completed the two tasks in the same order: the first without visualization support, and the second with visualization support. Design trade-offs and limitations are further discussed in Section 5.7.

The software project used for the evaluation was TeamMates, an open-source peer-evaluation system for education.[1] For reproducibility following commit version was used: `b24519a2af9e17b2bc9c025e87e4cf60009c425d`.

[1] https://teammatesv4.appspot.com/web/front/home

Following the tasks, a post-study questionnaire collected participants impressions and perspectives through both closed and open questions.

A preliminary pilot study with a single tester was conducted to refine the study design and identify potential technical or procedural issues, particularly regarding usability and setup experience.

The evaluation focused specifically on the SAD–Code visualization based on pre-generated gold-standard trace links between documentation and source files. To avoid negativ effects due to varying trace link accuracy, participants did not generate new links themselves.

A heuristic aggregation approach was applied: a directory-level visualization was shown only when all files within that directory were linked to the same documentation line. Consequently, the study evaluated both line-to-file and line-to-directory visualization interactions.

4.3. Study Process and Data Collection

This section describes the overall study procedure and the data collection process. To ensure transparency, reproducibility, and methodological rigor, all participants followed the same structured workflow, consisting of an introduction, two task phases, and a concluding questionnaire. The design integrates both qualitative and quantitative elements.

4.3.1. Introduction Script

To minimize bias and ensure reproducibility, a standardized introduction script was used for all participants. The script welcomed the participant, outlined the study's purpose and duration, explained the thesis goals, and obtained informed consent for audio recording. Afterwards, the participant was introduced to the software project and study procedure.

4.3.2. Tasks

Participants completed two main tasks designed to assess their ability to navigate and comprehend relationships within a software project:

Task 1 – CSV (without extension): Identify all code locations involved when creating a new course, from user input to storing the data in the datastore. Mark all relevant classes, methods, or files implementing this process.

Task 2 – *Trace-Viz* (with extension): Identify all code locations involved in verifying access control when a user sends a request to the system. Mark all relevant classes, methods, or files directly or indirectly implementing this functionality.

The study was structured in two phases. In the first phase, without the extension, participants completed a comprehension task using only conventional code navigation. In the second phase, with the extension, participants performed a similar task using the developed trace visualization extension. The extension available to the participants was commit `2c59f846a033782c140fe38abaed13b287396a7c`.

4.3.3. Questionnaire

Data was collected using audio recordings of the think-aloud sessions (with participant consent). In addition, a post-study questionnaire assessed usability, satisfaction, and perceived effectiveness. The questionnaire (see Appendix A.3) provided both quantitative and qualitative data, with a focus on user perception and subjective workload.

Following recommendations for qualitative data collection [12], the first round of qualitative analysis—based on the QBA coding method—was conducted after the first two participant sessions. This early analysis revealed the need for additional pre-study questions concerning the perceived difficulty of the two tasks. To address discrepancies between the pre-study assumptions and the experiences reported by the first two participants, the subsequent five sessions explicitly incorporated these additional questions. To limit the influence of growing familiarity with the visualization, a pre-task difficulty estimate was added: after the explanation of both tasks, participants were asked to rate the expected difficulty of each task before performing them.

This adjustment offered two advantages. First, it allowed the remaining sessions to focus more directly on the central research question: the effectiveness of the trace visualization extension. Second, it enabled the identification and control of potential error factors, even though such mid-study design changes inherently pose a challenge to data validity.

During the early coding phase, an additional methodological concern emerged. One participant noted that the first task appeared easier overall; this observation is plausible, as a learning effect between Task 1 and Task 2 cannot be fully avoided. To account for this potential bias, the post-study questionnaire includes the item: “How difficult did you perceive the task, and how long would you need to complete it?”. However, perceived difficulty after task completion may still be influenced by task order or accumulated knowledge.

To better distinguish the actual effectiveness of the extension from inherent differences in task complexity, an additional pre-study measure was therefore introduced. By asking participants to estimate the expected difficulty of each task in advance, the study design enables a direct comparison between expected and experienced difficulty. This, in turn, supports a more reliable interpretation of whether improvements in speed or accuracy can be attributed to the trace visualization extension itself rather than differences in task difficulty.

4.4. Data Analysis and Reporting

Two complementary approaches were applied to analyze the collected data. First, the questionnaire responses were examined item by item to identify recurring themes and trends in user feedback, and descriptive statistics were used where appropriate to summarize quantitative indicators. Second, the think-aloud recordings were transcribed using the tool UniScribe.[2] The resulting transcripts were analyzed using the Query-Based Analysis (QBA) method proposed by David L. Morgan (2023) [22]. QBA is a structured, AI-assisted framework for qualitative data analysis that employs generative language models, here in this study, ChatGPT 5.1, to support the systematic extraction and refinement of themes. Unlike traditional inductive coding procedures, which rely heavily on manual annotation, QBA begins with AI-generated thematic summaries that are iteratively refined in a controlled dialogue between researcher and model.

Morgan describes QBA as proceeding in three sequential stages. The first stage consists of broad, undirected queries intended to surface general patterns and high-level themes within the data. The second stage refines these preliminary insights by means of more targeted questions that explore subthemes and relate them explicitly to the research question. The third stage involves the examination of the original transcript to retrieve quotations that substantiate, nuance, or challenge the emerging interpretations. In this study, prompts are for broad inquiries *"What are the main topics covered in these documents?"*, more focused queries addressing the research question *"What are the main topics regarding my research question?"*, and evidence-oriented requests *"Give me quotes that support the results."*

The results were compared across participants to identify convergent and divergent themes, thereby enabling a more comprehensive interpretation of the collected data. This approach supported a structured analysis of developers reasoning processes, their interactions with the tool, and their perceptions of the usefulness of the visualization features during the think-aloud sessions.

4.5. Study subjects

Participants were recruited via institutional mailing lists, study groups, and personal developer contacts. A total of seven individuals took part in the study. Their backgrounds included three doctoral candidates, three master's students or graduates, and one industry developer with more than ten years of experience. All participants identified as male, with ages ranging from their twenties to thirties.

[2] https://www.uniscribe.co

5. Results and Discussion

In this chapter, the results of the study are presented and discussed. The chapter is structured according to the different types of data collected during the study, in the order in which they were obtained. It begins with the participants estimated task difficulty based solely on the task descriptions and a high-level architecture of the project. This initial data point serves as a reference to contextualize and compare the results gathered later in the study. Next, the artifacts (classes, methods, packages) identified as relevant for Tasks 1 and 2 are presented. This is followed by observations made during the completion of both tasks, which are described and discussed. Subsequently, the think-aloud transcripts are evaluated and interpreted. Finally, the results of the post-study questionnaire are summarized and discussed.

A total of seven participants took part in the study. All seven participants completed both tasks and filled out the post-survey questionnaire. Both tasks were conducted as a think-aloud study: participants were asked to verbalize everything they were thinking while working on the tasks. As expected, not all participants expressed their thoughts to the same extent; some needed occasional reminders to continue verbalizing their reasoning.

The study was conducted in German, as this was the mother tongue of all but one participant. Consequently, the questionnaire and instructions were also provided in German. For this chapter, the questions and answers have been translated or paraphrased into English. The original survey documents can be found in the appendix: the introduction script (A.1), which was used to reduce variance and provide all participants with the same starting point. The instructions for *trace-viz* (A.2), which were presented to participants before Task 2 to ensure comparable prior knowledge. Finally the post-survey questionnaire (A.3), which was used to analyze the research questions and to collect demographic information.

5.1. Estimated Task Difficulty

As described in the study design, two additional questions were added to the introduction script after the first two participants (A.1). These questions aimed to capture how difficult participants expected each task to be before working on them. Since, at this point, participants had only seen the task descriptions and a high-level architecture of the project, the ratings provide a baseline for later comparison with their actual experiences.

As shown in Figure 5.1, three out of five participants rated Task 1 as *hard*. One participant selected *middle*, and one selected *easy*. None considered the task to be very hard or very easy.

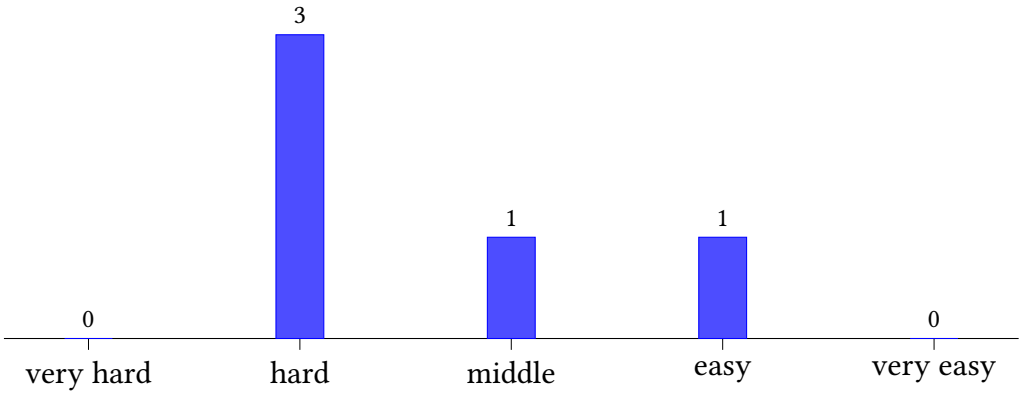


Figure 5.1.: How difficult do you consider **task 1** to be? (bar chart)

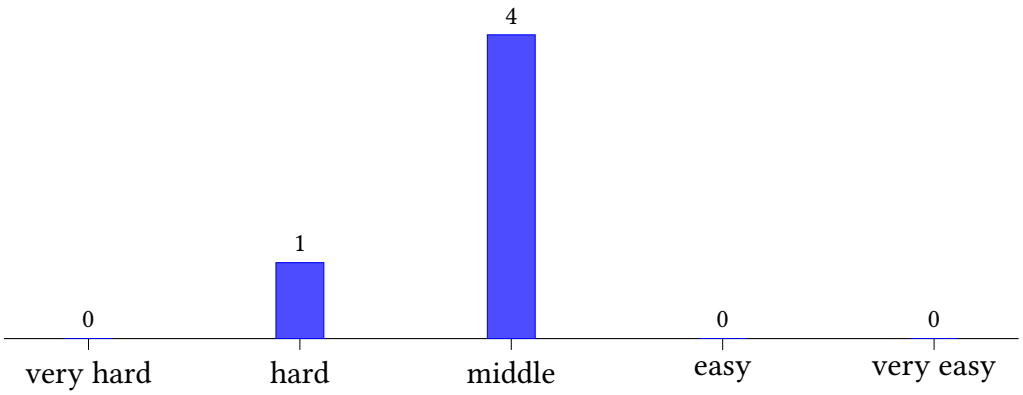


Figure 5.2.: How difficult do you consider **task 2** to be? (bar chart)

For Task 2 Figure 5.2 shows, four of the five participants selected *middle*, while one chose *hard*. Again, no participant rated the task as very hard or easy.

These baseline expectations are relevant for interpreting later results. For example, if participants initially predicted Task 1 to be harder, this shapes how we understand their later difficulties and navigation behaviour. Likewise, the fact that Task 2 was largely rated as "middle" to "hard" suggests that improvements in speed, confidence, or correctness observed later could more confidently be attributed to the visualization rather than differing task complexity. Overall, the estimated difficulty ratings serve as a useful reference point and indicate that participants approached the study with realistic expectations about the complexity of the tasks.

The estimated difficulty ratings show that participants expected both tasks to be moderately challenging, even before engaging with the code base. Task 1 was perceived as slightly more difficult than Task 2, which aligns with the qualitative feedback gathered later in the

study: Task 1 required participants to reconstruct a process (course creation) with no visual guidance, whereas Task 2 focused on access control with *trace-viz* guidance. While the Task 2 topic (access control) may be conceptually more familiar it is also not an individual feature (course creation) but a feature spanning throughout the entire project. Multiple participants noted this during the think-aloud phase and retrospectively revised their perception, stating that Task 2 was actually more difficult than they initially assumed. This shift in perceived difficulty is noteworthy: if Task 2 was inherently more complex yet was still experienced as easier when performed with visualization support, this provides an additional indication of the effectiveness of the *trace-viz* visualization.

5.2. Identified Artifacts

A further data point are the classes, methods, and files that participants identified as relevant during the two tasks.

Task 1: *Find all code locations involved in creating a new course—from user input to saving in the datastore. Mark all relevant classes, methods, or files that implement this process.*

Figure 5.3 shows the set of artifacts marked as relevant by the seven participants. The entries are sorted in descending order by the number of times they were selected. The most frequently identified artifacts were `CreateCourseAction.java` (5 participants), followed by `CoursesDb.java` (4 participants). Several other classes such as `Course.java`, `CourseCreateRequest.java`, and `CourseAttributes.java` were each identified by three participants.

These results reflect a reasonable spread: most of the participants identified the core workflow components (action handler, database access, and central entity classes), while peripheral or structural artifacts (e.g. `InstructorCoursesPage.java`, `WebAPIServlet.java`, or `DataBundle.java`) were only marked occasionally.

Task 2: *Find all code locations that are involved in checking access rights (access control) when a user sends a request to the system. Mark all relevant classes, methods, or files that implement this functionality directly or indirectly.*

Figure 5.4 presents the results for Task 2. The majority of users selected a small set of key classes. `AuthType.java` and `GateKeeper.java` were each selected by six participants, closely followed by `Action.java` (five participants) and `UnauthorizedAccessException.java` (four participants). A variety of additional artifacts such as `Action.checkAccessControl()`, `UserInfo.java`, or `OriginCheckFilter.java` were occasionally marked, reflecting indirect or contextual relevance.

Compared to Task 1, Task 2 produced a much clearer consensus on which artifacts were central. Participants heavily converged on a few well-defined components that implement the access-control decision path.

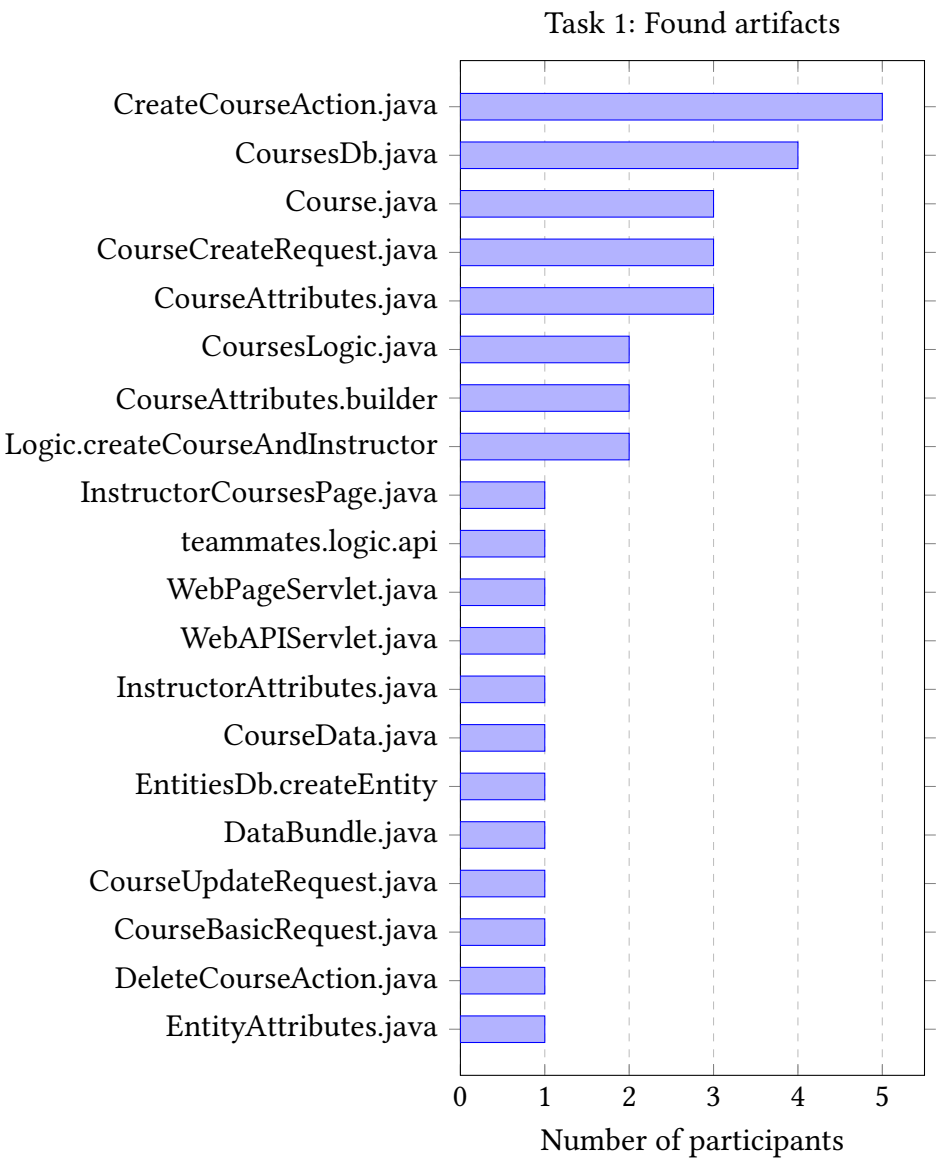


Figure 5.3.: Relevant artifacts found **task 1** (bar chart)

The artifact-selection patterns highlight important differences between the two tasks and provide insight into how participants navigated the system. Task 1 (course creation) produced a broader and more diverse set of identified artifacts. Without visualization, participants relied primarily on keyword search and manual navigation, leading to more heterogeneous results. Task 2 (access control) shows much stronger agreement. Most participants identified the same central classes (`GateKeeper`, `AuthType`, and `Action`). This suggests that, with the visualization support of *trace-viz*, relevant entry points became easier to spot, enabling participants to locate the key logic more consistently.

The difference between the two distributions indirectly illustrates the impact of *trace-viz*. In Task 1, performed *without visualization*, participants produced a more scattered set of

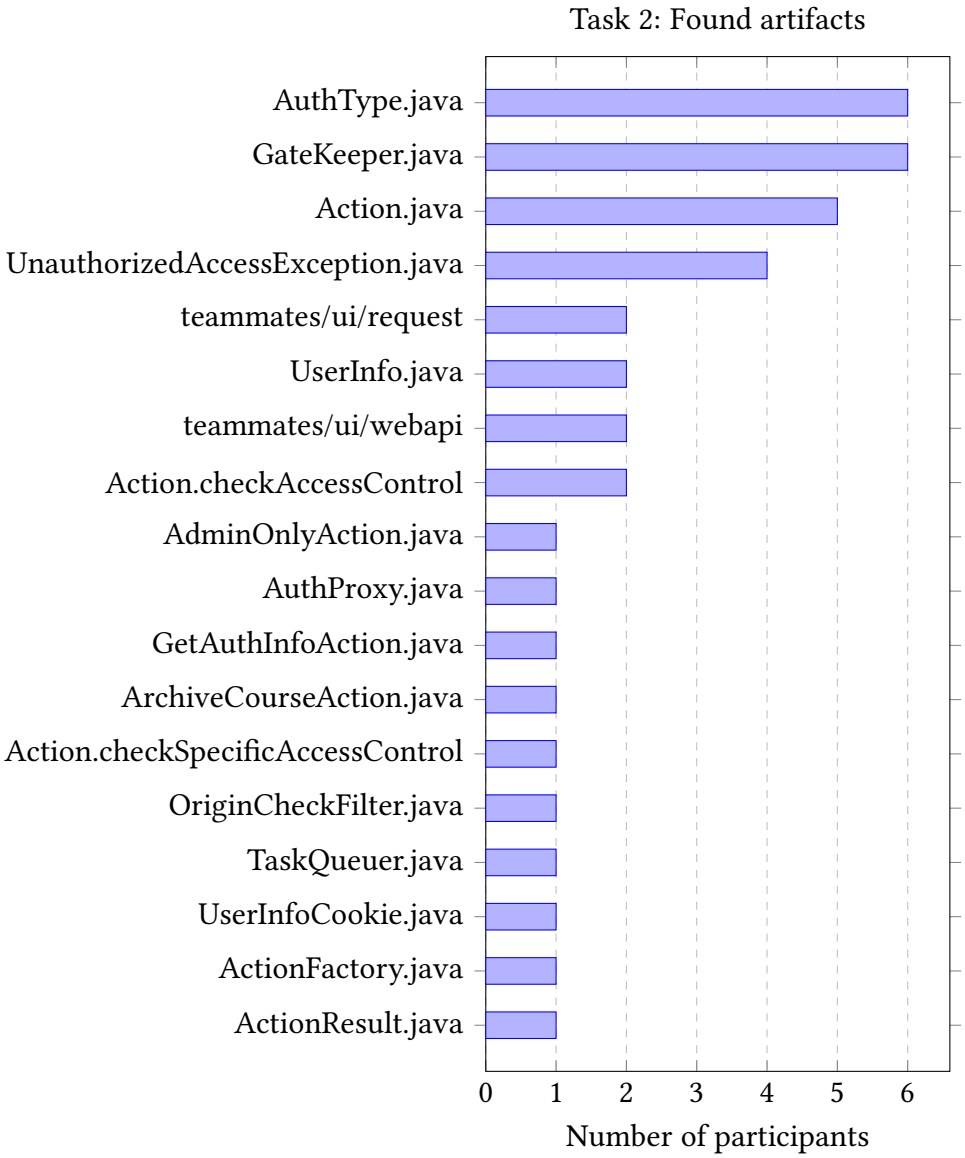


Figure 5.4.: Relevant artifacts found **task 2** (bar chart)

results. In Task 2, performed *with visualization*, participants converged on a clear subset of relevant elements, and required fewer navigation steps. This matches the qualitative think-aloud observations (e.g., frequent mentions of "click and click and click…" in Task 1 versus direct jumps in Task 2) and supports the later questionnaire findings that the visualization improved both efficiency and clarity.

5.3. Time Effort

A further data point concerns the time participants required to complete the two tasks. Each participant was given a maximum of ten minutes per task. Most participants used the full available time, while two participants finished earlier: one completed Task 2 in approximately seven minutes, and another declared Task 2 finished after about five minutes.

In addition to the actual time spent, participants were also asked to estimate how long they believed each task would take. These estimates provide a subjective measure of perceived workload and expected complexity.

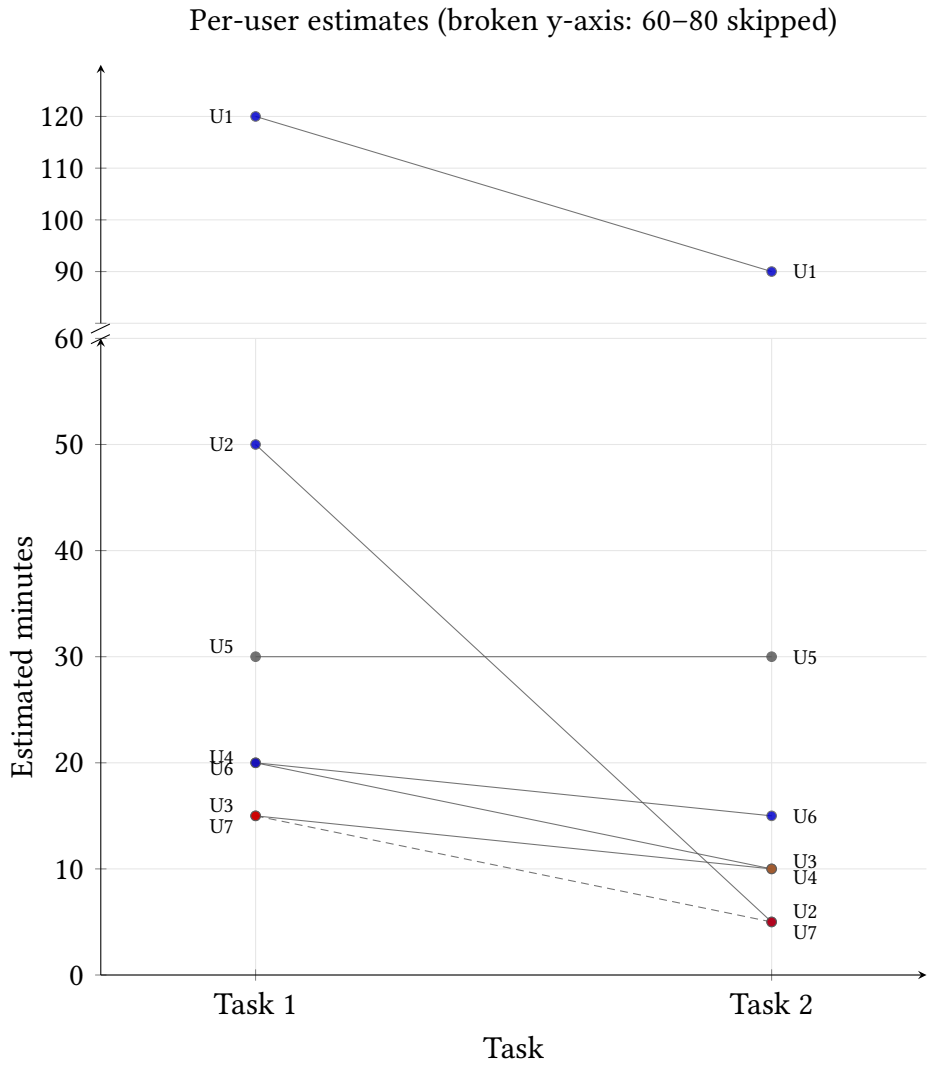


Figure 5.5.: Estimated time per task (slopegraph)

Figure 5.5 visualizes the individual estimates for both tasks using a slopegraph with a broken y-axis. Each line represents a single participant, connecting their estimated duration for Task 1 and Task 2. The break in the y-axis (between 60 and 80 minutes) is introduced to

clearly separate one outlier User 1 from the remaining estimates. Without this break, the outlier would compress the main cluster of values and reduce readability.

The majority of participants estimated both tasks to require between 5 and 30 minutes. User 1 provided substantially higher estimates (120 minutes for Task 1 and 90 minutes for Task 2), but the relative difference between their task ratings is still consistent with the rest of the participants: Task 2 was expected to be slightly quicker than Task 1.

Overall, the slopegraph shows a small but consistent downward trend from Task 1 to Task 2. Most participants assumed that Task 2 would take less time, even though—based on think-aloud feedback—some later realized that access control is conceptually broad and spans many parts of the system. This shift between expected and experienced difficulty mirrors earlier observations: the visualization in Task 2 made the task appear easier and faster, even though the underlying code structures were more widely distributed. The time estimates therefore reinforce the qualitative impression that *trace-viz* helped reduce perceived workload and increased participants confidence in navigating the system.

5.4. Navigation Effort

Another data point concerns the number of clicks required to navigate to the relevant files. This metric was not systematically collected for all participants, as navigation strategies varied: some participants relied heavily on the global project search, while others primarily navigated through the project tree view. Nevertheless, several indicative observations emerged.

Certain files, such as `DataBundle.java` and `CourseAttributes.java` required six clicks to reach from the project root when using the tree view. This depth resulted in repeated think-aloud comments such as *"click and click and click…"*, expressing frustration with the amount of navigation needed in Task 1. For many artifacts relevant to Task 1, the corresponding files were located at similar levels in the project hierarchy, amplifying the navigation effort without visualization support.

In contrast, during Task 2, participants were able to use the visualized trace links to jump directly to the relevant classes or packages. This drastically reduced the number of navigation steps and minimized the need to manually traverse the directory structure.

The navigation observations reveal a practical difference between working with and without visualization support. In Task 1—conducted without *trace-viz*—participants often relied on manual exploration of the project tree, resulting in long navigation paths and noticeable frustration. This aligns with the broader dispersion of artifacts identified in Task 1 and the think-aloud comments indicating uncertainty and repetitive tree traversal.

By comparison, the direct linking in Task 2 allowed participants to bypass the project hierarchy entirely. This not only reduced physical interaction effort (fewer clicks) but also lowered cognitive load: instead of searching, participants could follow explicit connections presented by the visualization. The reduced navigation friction likely contributed to the

more consistent artifact selection in Task 2 and to participants perception of the task as easier and faster, even though access control spans more components.

Overall, the navigation data provides an additional qualitative indication that *trace-viz* improved efficiency by simplifying movement through the codebase, supporting the findings from the questionnaire and think-aloud protocols.

5.5. Think-Aloud Observations

As the study was conducted as a think-aloud study and all seven participants agreed to the recording and analysis of the anonymized transcripts, additional insights could be gathered from their real-time problem-solving behaviour. For the analysis, the Query-Based Analysis (QBA) method proposed by David L. Morgan (2023) [22] was used on the transcripts. The transcripts also include participant's verbal reflections during the questionnaire phase, offering further context about their experiences.

Across all participants, Task 1 (CSV-only) was characterized by uncertainty, exploratory behaviour, and difficulty interpreting the trace links. Several participants explicitly struggled with the CSV format and the accompanying documentation. One participant described a trace entry as "beautifully unspecific", noting that the line "says exactly nothing" about which UI, logic, or storage components were involved in the course-creation path. Others similarly commented that the documentation "says nothing" beyond listing very broad categories such as UI, logic, and storage. The absence of concrete guidance, combined with the total number of trace links (around 7000), led several participants to describe Task 1 as "hard" or "very difficult" while working on it.

Without visualization support, participants resorted to ad-hoc heuristics to cope with the complexity. Common strategies included global keyword search (e.g., "course", "create", "access control"), scanning long lists of search results, and repeatedly navigating deep directory structures via the tree view. One participant attempted to infer relevant locations by searching for all usages of `UnauthorizedAccessException`, concluding that "wherever this exception is thrown, some evaluation of access rights must be happening," which produced more than 70 candidate files. Another participant, after opening "so many files", realized late in the task that a more effective strategy would have been to begin with the central action class and follow its dependencies, openly reflecting that their initial approach had been "wrong" and that the new strategy "would work nicer". These observations suggest that, without visualization, participants invested considerable effort into constructing their own mental model of the system and the trace links, and that this process was error-prone and cognitively demanding.

In Task 2, the introduction of *trace-viz* fundamentally changed both navigation behaviour and perceived difficulty. Participants frequently used the visual dots in the documentation as primary entry points and relied on the extension to jump directly from a documentation line to the corresponding file or directory. One participant remarked: "It's so easy – I just search for 'access', go to the line, and it immediately shows me the file," adding that they

"really liked it." The directory heuristic, which collapses multiple file-level trace links into a single directory-level link, was also explicitly appreciated, as it allowed participants to reach entire groups of relevant classes with one action instead of drilling through nested packages. In several cases, participants abandoned earlier exploratory strategies (such as scanning the project tree) and shifted to using the visualization as their primary navigation mechanism.

The think-aloud data also show that visualization improved not only speed but also conceptual understanding. For access control in Task 2, multiple participants quickly converged on the same core classes (`GateKeeper`, `AuthType`, `Action`) and articulated the layered structure of access checks in the web API and action layer. One participant, for example, used the visual trace links to follow the description "the web API is protected by two layers of access control checks" and then reasoned through where `checkAccessControl` and `checkSpecificAccessControl` are implemented and how they are inherited across different action classes. However, participants also pointed out limitations of the visualization. For instance, one participant noticed that a trace-link referenced in line 89 was missing entirely. Others remarked that if a trace-link to `GateKeeper` had been available, they "could just click it and immediately find the files," emphasizing that incomplete or missing trace links can directly reduce the effectiveness of the visualization and interrupt the otherwise smooth navigation experience.

Finally, several participants revisited their initial assumptions about task difficulty. Although course creation (Task 1) represents a single functional feature, multiple participants described it as "a big task" because the relevant artifacts were widely scattered and difficult to locate without guidance. In contrast, access control (Task 2) was recognized as a cross-cutting concern that "touches almost everything," yet was still experienced as more manageable due to the structural guidance provided by *trace-viz*. One participant reflected that because access control appears everywhere, "everything is relevant," yet the visualization allowed them to identify the correct entry points quickly. This re-evaluation aligns with the questionnaire findings: although Task 2 spans more components and was initially rated as "middle" to "hard," it was later perceived as easier when performed with visualization support.

Overall, the think-aloud observations provide qualitative evidence that the integrated VS Code extension reduced navigation effort, supported the formation of more accurate mental models, and improved perceived efficiency. Visualization not only guided participants to relevant artifacts more directly but also helped them understand relationships between documentation and code more effectively than raw CSV trace links alone.

5.6. Questionnaire Results

The questionnaire was completed by participants after they had finished both tasks. It was structured into several thematic sections and included a mix of Likert-scale items, closed questions, open-ended questions, and demographic items.

1. General Understanding and Clarity

This section was designed to assess how well participants understood the study instructions, the purpose of the tasks, and the overall clarity of the materials. It evaluates whether the instructions were comprehensible, whether any confusion occurred during task execution, and how clearly participants perceived the study's objectives.

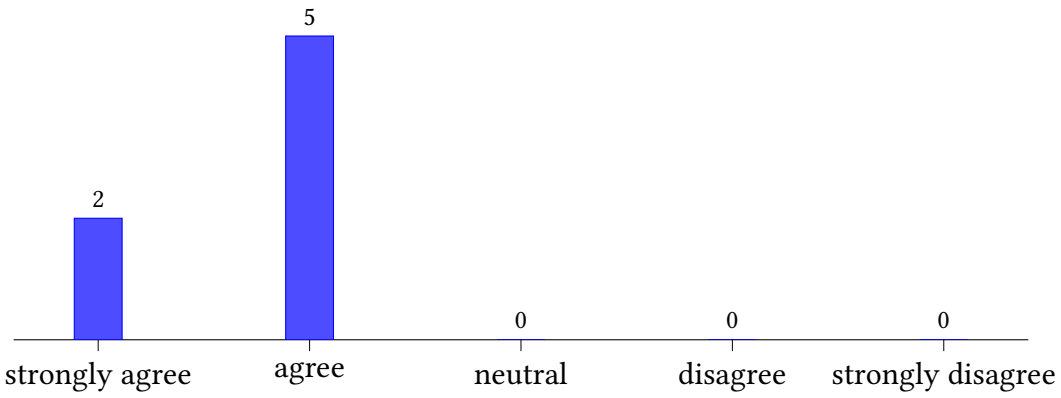


Figure 5.6.: Is the task clear to you overall? (bar chart)

As shown in Figure 5.6, all participants either agreed or strongly agreed that the tasks were clear. Six of the seven participants additionally answered "no" to the question "Did you have any difficulty understanding the task at any point?" This indicates a consistently high level of clarity in the task formulation.

Responses to the open question *"How clear was the purpose of the study to you, or what was expected of you?"* further reinforce this impression. All but one participant stated that the purpose was clear. The remaining participant suggested that a brief definition of trace links would have been helpful.

Overall, the clarity-related responses demonstrate that participants were able to understand the instructions and the goals of the study without substantial confusion. This is important for interpreting subsequent results: if participants clearly understood what they were expected to do, then differences in task performance or artifact identification can more confidently be attributed to the experimental conditions (e.g., presence or absence of visualization) rather than misunderstandings about the task itself.

The single comment requesting a definition of trace links highlights a minor gap in prior knowledge, but does not appear to have affected the general task comprehension. Instead, it underscores that while the concept may not be universally familiar, the actual task execution remained sufficiently clear for all participants.

2. Task Procedure and Use of Trace links

This section aimed to capture how participants approached the tasks, which strategies they employed during problem-solving, and how they made use of the provided trace links. It examines their decision-making processes, the techniques they found helpful for identifying

relevant information, and the differences they perceived when working with and without visualized trace links.

Six of the seven participants reported noticing a clear difference between Task 1 and Task 2. In response to the question "If so, what were the differences?", participants highlighted several aspects of the visualization that supported their work in Task 2: the grouping of trace links by package, the ability to jump directly to related files, and the overall ease of locating relevant code elements. These features reduced the need for manual exploration of the project tree and made the underlying structure more transparent.

Across both tasks, the primary search strategy was the use of keyword search (e.g., `cmd+f`). All participants relied on this method to some extent. In Task 1, keyword search often served as the main navigation tool due to the absence of visualization support. In Task 2, it was used more selectively, typically in combination with the trace-link jumps provided by the visualization.

The responses indicate that participants adapted their strategies depending on the available support. Without visualization (Task 1), they primarily relied on keyword search and manual tree navigation, which often led to repetitive and exploratory behavior. With visualization (Task 2), participants shifted toward a more targeted approach, using the direct trace-link jumps to navigate efficiently between relevant artifacts.

The fact that six participants explicitly recognized and articulated these differences suggests that the visualization meaningfully influenced their workflow. Rather than changing what they searched for, it changed how they navigated the codebase. This reduction in navigation overhead aligns with improvements observed in artifact identification accuracy and with participants later questionnaire responses indicating that the visualization supported their comprehension process.

Overall, these insights reinforce the interpretation that *trace-viz* not only provided structural guidance but also encouraged more intentional and efficient search strategies during Task 2.

3. Comparison of Both Tasks

This section compares participants experiences and perceived difficulty across the two task variants one completed without visualization and one supported by the *trace-viz* visualization. It examines which version participants found easier, how they justified their preference, and to what extent the visualization influenced their performance, understanding, and overall task experience.

In response to the question *"Which of the two tasks (with or without visualization) did you find easier?"*, six out of seven participants selected *"with visualization."* One participant chose *"same difficulty,"* explaining that both tasks were challenging overall. The participants who preferred the visualization emphasized that it added structure to the search results, provided a more comprehensive overview, and made relevant locations easier to identify. Statements such as "You have a much more comprehensive overview" and "The tool is useful" reflect a consistent perception of increased clarity and reduced effort.

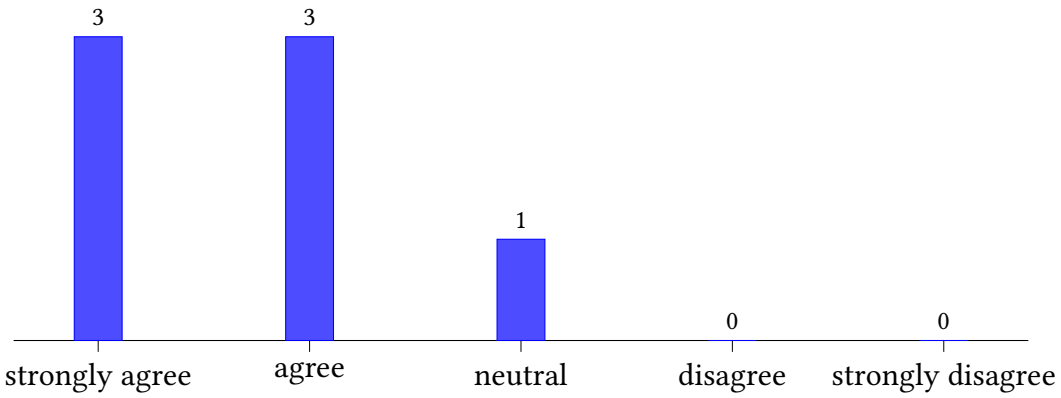


Figure 5.7.: Visualizing the trace links in the second task was helpful? (bar chart)

Figure 5.7 presents the responses to the question of whether visualizing the trace links in the second task was helpful. Three participants selected strongly agree, another three selected agree, and one participant chose neutral. No responses were recorded for disagree or strongly disagree.

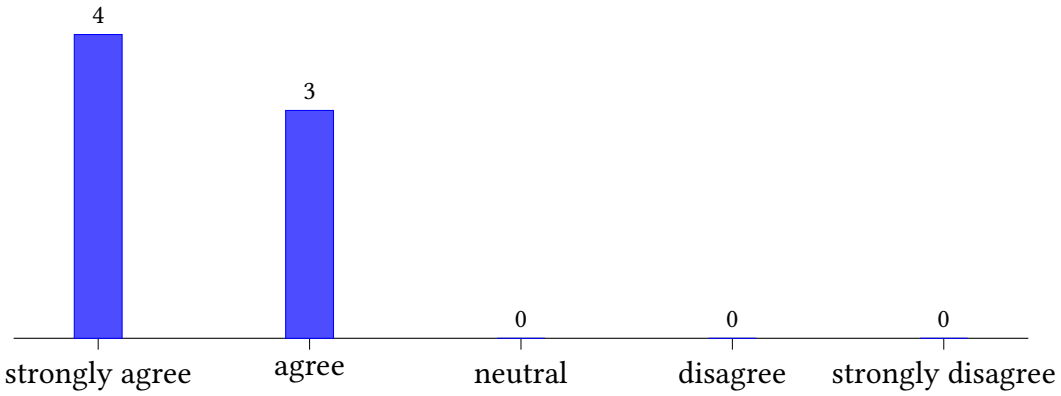


Figure 5.8.: The visualization supported your search or comprehension process? (bar chart)

Figure 5.8 shows the responses to whether the visualization supported the participants search or comprehension process. Four participants selected strongly agree and three participants selected agree. No participant chose neutral, disagree, or strongly disagree.

Both Figure 5.7 and Figure 5.8 therefore report exclusively positive or neutral ratings regarding the extent to which the visualization influenced the participants performance and understanding.

These results provide strong evidence that the visualization meaningfully improved participants task experience. The fact that six participants explicitly stated that the visualization

made the task easier (and none indicated the opposite) suggests that the added structure, direct navigation, and visual grouping were perceived as highly beneficial.

The consistent agreement in Figure 5.7 and Figure 5.8 complements earlier findings from the think-aloud data: participants in Task 2 spent less time navigating, identified relevant artifacts more consistently, and expressed less uncertainty. The visualization appears to have reduced both cognitive load (by clarifying the system structure) and mechanical effort (by reducing navigation steps).

Interestingly, one participant still perceived both tasks as equally difficult. This response underscores that perceived difficulty is multifaceted. While visualization reduces search effort, underlying code complexity, particularly in access control, may still contribute to overall challenge. Nevertheless, even this participant agreed that the visualization itself was helpful.

Overall, the comparative results illustrate that *trace-viz* effectively supported participants in locating, understanding, and navigating relevant artifacts—key aspects of developer comprehension tasks.

4. Effort and Usability

This section aimed to assess the perceived workload, time requirements, and overall usability of the tasks and the visualization tool. It investigates how much time participants felt they needed, whether they experienced uncertainty during the process.

The estimated time per task has already been analyzed in Figure 5.5. In addition to providing estimates, participants were asked whether they felt they had enough time to complete both tasks.

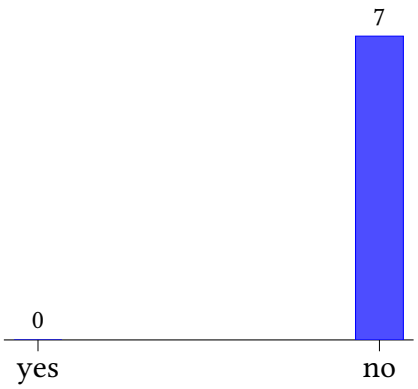


Figure 5.9.: Do you feel you have enough time to complete both tasks? (bar chart)

As shown in Figure 5.11, all seven participants indicated that they did not feel they had enough time to complete both tasks. This outcome is unsurprising, given that the tasks

required navigating an unfamiliar codebase and that the study design intentionally limited time to observe participants' prioritization and navigation behavior under moderate pressure.

Furthermore, all but one participant reported feeling confident about how to proceed during the tasks. The single participant who expressed uncertainty experienced this primarily during Task 1, where no visualization support was available.

The uniformly negative responses regarding available time indicate that participants perceived both tasks as demanding. Despite this, almost all participants felt confident about how to proceed, suggesting that time pressure stemmed primarily from navigation effort rather than task comprehension. This aligns with earlier think-aloud observations: in Task 1, manual navigation was slow and repetitive, while in Task 2 the visualization reduced mechanical effort and clarified the structure. Overall, the workload-related responses support the interpretation that *trace-viz* helped reduce perceived time pressure by enabling more efficient navigation in the codebase.

5. User Experience and Preference

This section aimed to evaluate participants subjective experience with the visualization tool, including its intuitiveness, perceived usefulness, and potential applicability in real-world work contexts. It also explores which visualization features participants preferred, which aspects they found confusing or unnecessary, and whether they would choose to use such a tool in their professional practice.

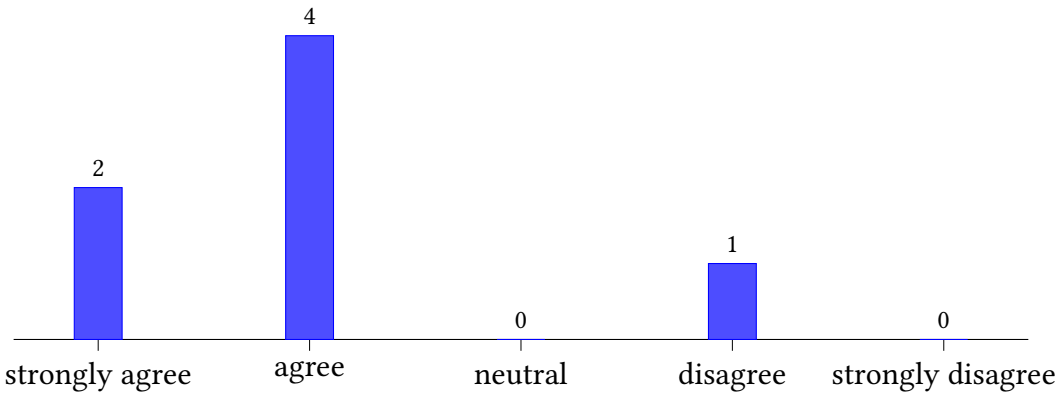


Figure 5.10.: Was the visualization intuitive for you to use? (bar chart)

Participants highlighted several features as particularly helpful, including the visual dots representing trace links, the grouping of related elements, and the ability to jump directly to a file. At the same time, some participants found the visualization of directory structures confusing or less useful.

The responses indicate that the visualization was generally perceived as intuitive and helpful, although not uniformly so. Six participants agreed or strongly agreed that the tool was

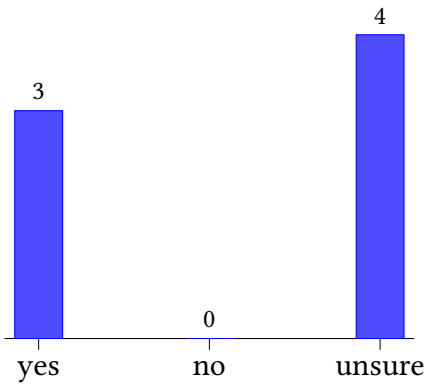


Figure 5.11.: Would you use such a visualization tool in a real work context? (bar chart)

intuitive, while one participant disagreed, suggesting that usability may depend on prior experience or individual work habits. Regarding real-world applicability, three participants stated they would use such a tool, and four were unsure. Notably, none rejected the idea. This hesitation likely reflects that the tool was demonstrated only in a controlled study setting and may require deeper integration or familiarity before participants can confidently assess its long-term value.

One participant also remarked that this type of task—exploring an unfamiliar codebase without prior guidance—does not reflect their *usual* work context. However, they could imagine the visualization being particularly useful in onboarding scenarios or when navigating large codebases where no guidance can be provided, such as in extensive open-source projects.

Overall, the results show clear potential for practical application, tempered by a need for refinement in areas such as directory visualization and onboarding clarity.

6. Perception and Suggestions

This section aimed to capture participants overall perception of the visualization's impact on their efficiency and accuracy, as well as gather concrete improvement suggestions. It focuses on how participants evaluated the effectiveness of the visualization, which enhancements they recommended for the tool or study procedure, and any additional observations they found relevant during the tasks.

As shown in Figure 5.12, participants responded overwhelmingly positively regarding the visualization's effect on efficiency and accuracy. There are eight responses from seven participants because one participant distinguished between efficiency (strongly agree) and accuracy (agree). This differentiation suggests that participants were actively reflecting on specific aspects of the visualization's benefits.

Participants provided suggestions for improvements, including individual preferences such as making the dots larger, introducing shortcuts, or enabling deeper trace-link navigation—including links to individual methods. The following question, *"Would you like to add*

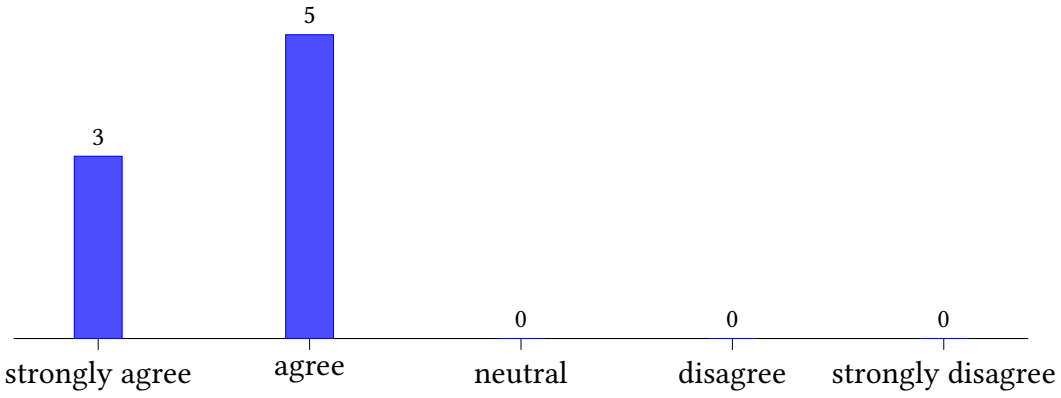


Figure 5.12.: Do you feel that visualization has improved your efficiency or accuracy? (bar chart)

anything else that caught your attention during processing?", raised the issue that the trace links also have limitations. For example, in line 89 the element GateKeeper is mentioned, but no corresponding trace-link exists.

Overall, participants were relatively neutral regarding their familiarity with trace links, and most were not familiar with the project TeamMates (with one participant indicating slight academic familiarity).

The consistently positive ratings indicate that participants perceived the visualization as a meaningful improvement to their efficiency and accuracy when navigating the codebase. The concrete suggestions they provided focus primarily on usability refinements and expanded functionality rather than fundamental issues, suggesting overall acceptance of the concept. Notably, the mention of missing trace links highlights limitations in the underlying traceability approach, which could directly affect user trust and comprehension. These insights point toward clear directions for future improvement: enhancing completeness of trace-link generation and extending navigation depth to method-level precision.

5.7. Limitations and Threats to validity

Several limitations, such as small sample size and the homogeneous nature limit this study. Furthermore potential biases by the researcher and the sequence of the tasks must be considered.

All participants were male and came from similar academic or professional backgrounds. The nature of the tasks required a computer science and programming background.

The study used a fixed project and fixed tasks, which limits generalizability. The order of conditions was also fixed: participants always performed the task without the extension first, introducing possible learning effects.

The small sample size especially limits the statistical reliability. Therfore, these questions can only be a start for a broader and more reliable study.

Finally, as the tool's developer, the evaluator may have unintentionally influenced participants during setup or explanation. Future studies should randomize task order. The participant diversity has to be increased. Furthermore, multiple projects will be able to enhance external validity.

6. Conclusion

This thesis examined how visualization of traceability information can support software developers in navigating and comprehending complex systems. Although modern Trace Link Recovery approaches such as ARDoCo and LiSSA provide increasingly powerful mechanisms to automatically generate trace links between documentation, models, and source code, their practical usage remains limited by a lack of integration into developers daily workflows. This gap was addressed in this work with *trace-viz*, a Visual Studio Code extension that brings trace links directly into the familiar development environment and enables their exploration directly within the IDE through integrated, developer-oriented visualization mechanisms.

The extension was developed using a modular, layered architecture that separates presentation logic, application coordination, domain services, and external integrations. This structure allowed the tool to support several traceability approaches, including CSV-based imports, REST-based communication with ARDoCo, and execution of LiSSA through a local JAR. The visualization mechanisms were designed to embed traceability information into the code editor in a minimally intrusive yet continuously accessible manner. Through colored gutter markers, hover-based annotations, and quick navigation options, the extension seeks to reduce cognitive load, shorten navigation paths, and improve the discoverability of relevant artifacts.

To evaluate the impact of the extension, a think-aloud study grounded in the Goal Question Metric (GQM) framework was conducted. With the overarching research question: To what extent does an integrated Visual Studio Code extension for guided exploration and visualization of trace links improve developers effectiveness and efficiency in understanding and navigating relationships between documentation, models, and code, compared to having access to the same trace links in a non-visualized, unstructured format? Participants were asked to solve two comprehension tasks in a realistic and unfamiliar codebase, one using only trace links ins CSV format and one aided by the visualization. The collected data, which included audio transcripts, navigation observations, questionnaire responses, and artifact selections, consistently demonstrated that the visualization improved both efficiency and effectiveness. Without visualization, participants relied heavily on keyword search and manual traversal of the project structure, which led to scattered exploration patterns, higher navigation effort, and greater uncertainty. In contrast, with *trace-viz*, they were able to identify relevant artifacts more consistently and to move between them with fewer interactions. The dot-based cues in the documentation acted as stable entry points into the code, and the ability to jump directly to files or directories substantially reduced the mechanical effort of exploration.

Participants also reported a notable improvement in their confidence while working with the visualized trace links. Even though the second task, which focused on access control, was conceptually broader and touched more components of the system than the first, it was generally perceived as easier when supported by *trace-viz*. This contrast suggests that the visualization not only accelerates navigation but also provides a structural scaffold that supports the formation of more accurate mental models of the system. Qualitative insights from the think-aloud data further revealed that participants were often able to articulate system relationships more clearly when guided by the visualization.

At the same time, the evaluation highlighted several limitations that shape the interpretation of the findings. The usefulness of the visualization depends directly on the completeness and accuracy of the underlying trace links; missing connections occasionally interrupted the otherwise smooth navigation experience. The study design, which always presented the CSV-based task first, may have introduced minor learning effects, and the sample size and participant demographics limit generalizability. Certain visualization elements, particularly directory-level representations, were perceived as less intuitive and would benefit from refinement. These limitations do not diminish the overall positive results but point toward opportunities for further work.

Future technical work should focus on extending the extension with additional traceability approaches and more advanced visualization modes. Furthermore, several usability improvements suggested in the user study, such as adding keyboard shortcuts should be implemented. These enhancements would increase efficiency, reduce navigation effort, and make the extension more adaptable for different developer workflows. From a methodological standpoint, future evaluations should involve a larger and more diverse sample to improve the robustness and generalizability of the results. Additional tasks and alternative task types should be incorporated to assess the tool across a broader range of comprehension and navigation scenarios. Moreover, changing or randomizing the task order would help account for learning effects that may have influenced the current study, ensuring a clearer separation between tool impact and task complexity. These refinements would allow for more reliable conclusions about the effectiveness of the visualization approach.

The findings of this thesis indicate that embedding traceability information directly into the coding environment can meaningfully support common developer activities such as locating relevant artifacts, understanding system structure, and navigating unfamiliar codebases. The study suggests that interactive visualization can help bridge the gap between automated trace link generation and its practical application in everyday work. While additional features, broader evaluations, and further integration with traceability frameworks would strengthen the extension, the results already demonstrate the potential of such tools to enhance software comprehension and maintenance.

In sum, *trace-viz* provides a concrete demonstration of how traceability research can be translated into developer-oriented practice. By situating trace links within the IDE and presenting them in an accessible, context-sensitive form, the extension contributes to making traceability not merely a theoretical quality assurance mechanism but a practical tool for guiding software exploration. Future investigations, involving larger studies, diverse codebases, and refined visualization techniques, have the potential to expand this line of

research further. Nonetheless, the present work shows that visualization can significantly strengthen the usability of traceability information and offers a promising foundation for integrating automated trace link recovery into real-world development processes.

Bibliography

[1] G. Antoniol et al. “Recovering traceability links between code and documentation”. In: *IEEE Transactions on Software Engineering* 28.10 (2002), pp. 970–983. DOI: 10.1109/TSE.2002.1041053.

[2] P. Arkley and S. Riddle. “Overcoming the traceability benefit problem”. In: *13th IEEE International Conference on Requirements Engineering (RE’05)*. 2005, pp. 385–389. DOI: 10.1109/RE.2005.49.

[3] Thazin Win Win Aung, Huan Huo, and Yulei Sui. “Interactive Traceability Links Visualization using Hierarchical Trace Map”. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 367–369. DOI: 10.1109/ICSME.2019.00059.

[4] Victor R Basili, Gianluigi Caldiera, and H Dieter Rombach. “THE GOAL QUESTION METRIC APPROACH”. en. In: ().

[5] Xiaofan Chen, John Hosking, and John Grundy. “Visualizing Traceability Links between Source Code and Documentation”. In: Sept. 2012. DOI: 10.1109/VLHCC.2012.6344496.

[6] Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman, eds. *Software and Systems Traceability*. London: Springer London, 2012. DOI: 10.1007/978-1-4471-2239-5. URL: https://doi.org/10.1007/978-1-4471-2239-5.

[7] Dassault Systèmes. *CATIA Reqtify - Requirements Traceability Tool*. https://www.3ds.com/products/catia/reqtify. Accessed: 2025-07-21. 2025. URL: https://www.3ds.com/products/catia/reqtify.

[8] Dominik Fuchß et al. “Beyond Retrieval: A Study of Using LLM Ensembles for Candidate Filtering in Requirements Traceability”. In: *2025 IEEE 33rd International Requirements Engineering Conference Workshops (REW)*. 2025, pp. 5–12. DOI: 10.1109/REW66121.2025.00006.

[9] Dominik Fuchß et al. “Enabling Architecture Traceability by LLM-based Architecture Component Name Extraction”. In: *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*. 22nd IEEE International Conference on Software Architecture. ICSA 2025 (Odense, Denmark, Mar. 31–Apr. 4, 2025). Institute of Electrical and Electronics Engineers (IEEE), Apr. 2025. DOI: 10.1109/ICSA65012.2025.00011.

[10] Dominik Fuchß et al. *LiSSA – LLM/RAG-based Traceability Link Recovery*. https://ardoco.de/approaches/lissa/. Accessed: 2025-11-21. 2025.

[11] Dominik Fuchß et al. “LiSSA: Toward Generic Traceability Link Recovery through Retrieval-Augmented Generation”. In: *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. ICSE ’25. Ottawa, Canada: Institute of Electrical and Electronics Engineers (IEEE), May 2025. DOI: 10.1109/ICSE55347.2025.00186.

[12] Corrine Glesne and Alan Peshkin. *Becoming qualitative researchers : an introduction*. White Plains, NY: Longman, 1992.

[13] Tobias Hey et al. “Requirements Traceability Link Recovery via Retrieval-Augmented Generation”. In: *Requirements Engineering: Foundation for Software Quality*. Cham: Springer, Apr. 2025. DOI: 10.1007/978-3-031-88531-0_27.

[14] Jan Keim et al. “Detecting Inconsistencies in Software Architecture Documentation Using Traceability Link Recovery”. In: *2023 IEEE 20th International Conference on Software Architecture (ICSA)*. 2023, pp. 141–152. DOI: 10.1109/ICSA56044.2023.00021.

[15] Jan Keim et al. “Detecting Inconsistencies in Software Architecture Documentation Using Traceability Link Recovery”. In: *2023 IEEE 20th International Conference on Software Architecture (ICSA)*. 2023, pp. 141–152. DOI: 10.1109/ICSA56044.2023.00021.

[16] Jan Keim et al. “Recovering Trace Links Between Software Documentation And Code”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE ’24. Lisbon, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3639130. URL: https://doi.org/10.1145/3597503.3639130.

[17] Jan Keim et al. “Trace Link Recovery for Software Architecture Documentation”. In: *Software Architecture*. Ed. by Stefan Biffl et al. Cham: Springer International Publishing, 2021, pp. 101–116. ISBN: 978-3-030-86044-8. DOI: 10.1007/978-3-030-86044-8_7.

[18] Stefan Kugele and Daniel Antkowiak. “Visualization of Trace Links and Change Impact Analysis”. In: *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*. 2016, pp. 165–169. DOI: 10.1109/REW.2016.039.

[19] Brian Di Loreto. *AtomicViz: Atomic Design System Visualizer*. https://github.com/briandiloreto/AtomicViz. Accessed: 2025-07-21. 2025. URL: https://github.com/briandiloreto/AtomicViz.

[20] Microsoft Corporation. *UX Guidelines | Visual Studio Code Extension API*. https://code.visualstudio.com/api/ux-guidelines/overview. Accessed: 2025-11-21. 2025.

[21] KASTEL Institute of Information Security Modelling for Continuous Software Engineering (MCSE) group and KIT Dependability. *ARDoCo – Automating Requirements and Documentation Comprehension*. https://ardoco.de. Accessed: 2025-11-21. 2025.

[22] David L. Morgan. “Query-Based Analysis: A Strategy for Analyzing Qualitative Data Using ChatGPT”. In: *Qualitative Health Research* 0.0 (0). PMID: 40481623, p. 10497323251321712. DOI: 10.1177/10497323251321712. eprint: https://doi.org/10.1177/10497323251321712. URL: https://doi.org/10.1177/10497323251321712.

[23] Ernesto Panadero, Leire Pinedo, and Javier Fernández Ruiz. “Unleashing think-aloud data to investigate self-assessment: Quantitative and qualitative approaches”. In: *Learning and Instruction* 95 (2025), p. 102031. ISSN: 0959-4752. DOI: https://doi.org/10.1016/j.learninstruc.2024.102031. URL: https://www.sciencedirect.com/science/article/pii/S0959475224001580.

[24] W.J.P. Van Ravensteijn. “Visual traceability across dynamic ordered hierarchies”. Master’s thesis. Eindhoven University of Technology, Aug. 2011.

[25] Joseph Sobolewski. *CodeGraphy: Code Graph Generation Tool*. https://github.com/joesobo/CodeGraphy. Accessed: 2025-07-21. 2025. URL: https://github.com/joesobo/CodeGraphy.

A. Appendix

- Github link to *trace-viz*: `https://github.com/ardoco/traceviz`

A.1. trace-viz Introduction Script

“trace-viz” Introduction Script – Think-Aloud-Study – DE

Herzlichen Willkommen! Vielen Dank das Sie sich die Zeit nehmen um an meiner Think-Aloud-Study für meine Bachelorarbeit teilnehmen. Die Studie dauert ungefähr 30-40 min.

Diese Bachelorarbeit stellt die Entwicklung einer Visual Studio Code Erweiterung zur interaktiven Exploration von Trace Links in Softwareprojekten vor. Die Erweiterung integriert generierte Trace Links von ArDoCo und LiSSA und visualisiert diese direkt in der Entwicklungsumgebung. Entwickler können Beziehungen zwischen Dokumentation, Modellen und Code in verschiedenen Ansichten erkunden. Ziel ist es, das Verständnis von Softwaresystemen zu verbessern und typische Aufgaben zu erleichtern, ohne den Arbeitsfluss zu unterbrechen. Die Wirksamkeit wird durch eine Think-Aloud-Nutzerstudie evaluiert.

Wären Sie damit einverstanden das ich den Think-Aloud Teil aufzeichne. Die Aufnahmen dienen ausschließlich zur Auswertung und werden Transkribiert und Anonymisiert und im Anschluss wieder gelöscht.

<Antwort Teilnehmer>

Sie bekommen gleich zwei Aufgaben gestellt. Eine die Sie mit Trace-Links als CSV Datei Lösen können. Eine die Sie mit Trace-Links im visualisieretem Format durch die VSCode Extension trace-viz visulaisiert bekommen.

<Projekt Übersicht TEAMMATES>

<Erläuterung beider Aufgaben>

Wie schätzen Sie die Schwierigkeit von **Aufgabe 1** ein?

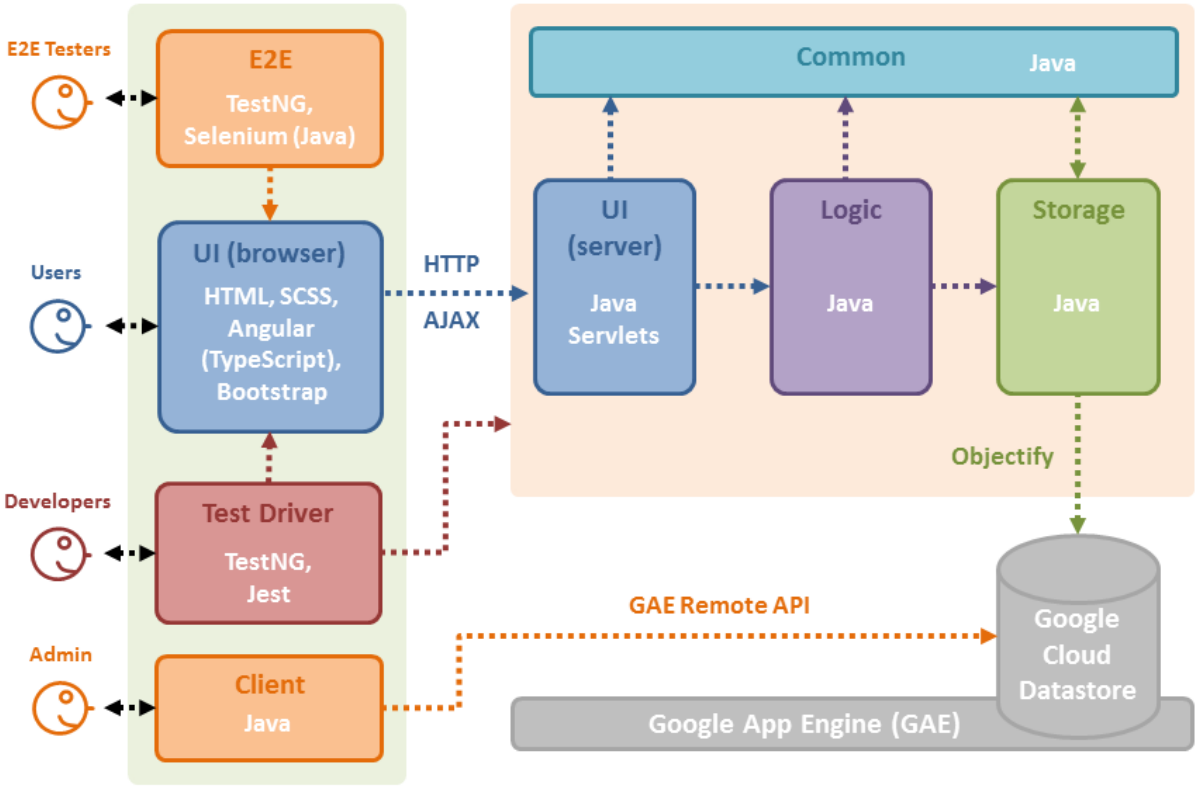
☐	☐	☐	☐	☐
Sehr schwer	Schwer	Mittel	Leicht	Sehr leicht

Wie schätzen Sie die Schwierigkeit von **Aufgabe 2** ein?

☐	☐	☐	☐	☐
Sehr schwer	Schwer	Mittel	Leicht	Sehr leicht

Wir arbeiten mit dem Projekt TEAMMATES https://teammatesv4.appspot.com/web/front/home
Student peer evaluations/feedback, shareable instructor comments, and more...

Hier eine Highlevel-Architektur Übersicht:



Quelle: https://github.com/ardoco/teammates/blob/0ad6268baad33e723a280deb30fcb0f3329faeb1/docs/images/highlevelArchitecture.png

Aufgabe 1 - CSV

Finden Sie alle Codestellen, die beim **Anlegen eines neuen Kurses** beteiligt sind – von der Benutzereingabe bis zum Speichern im Datastore. Markieren Sie alle relevanten Klassen, Methoden oder Dateien, die diesen Prozess implementieren.

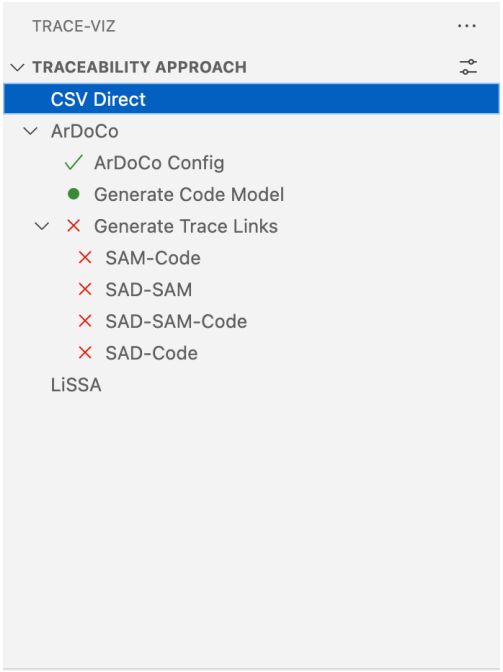
Aufgabe 2 – trace-viz

Finden Sie alle Codestellen, die an der **Prüfung von Zugriffsrechten (Access Control)** beteiligt sind, wenn ein Nutzer eine Anfrage an das System sendet. Markieren Sie alle relevanten Klassen, Methoden oder Dateien, die diese Funktionalität direkt oder indirekt umsetzen.

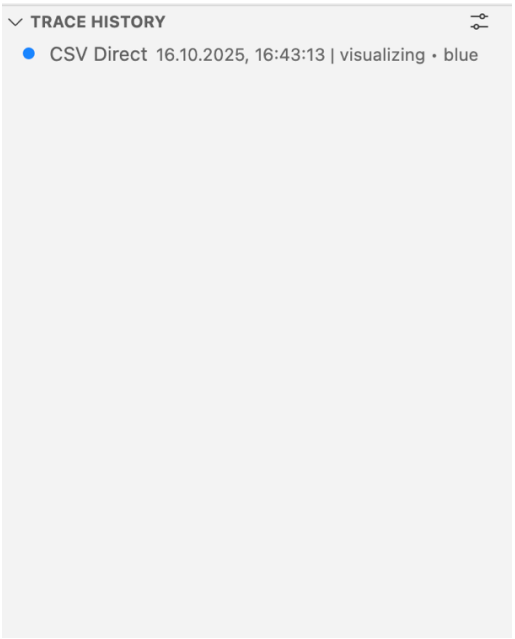
A.2. trace-viz Instructions

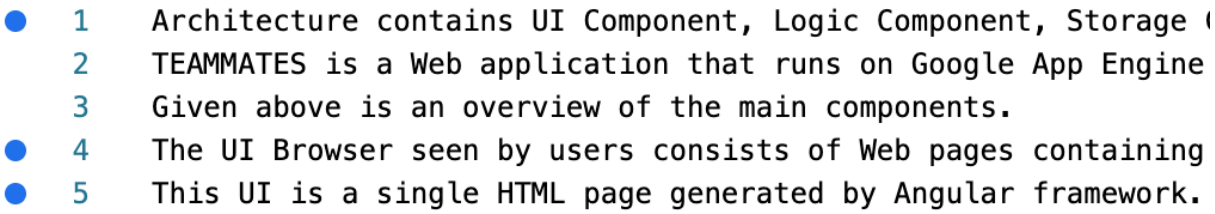
Anleitung “trace-viz”

Traceability Approches:



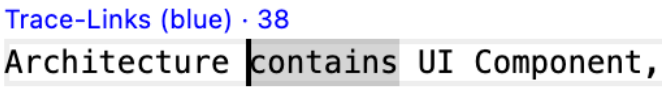
Trace History:



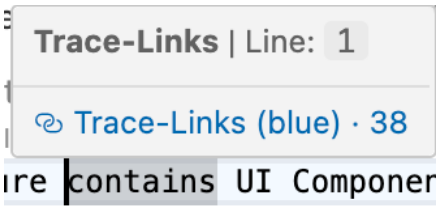


Punkte zeigen an das es Trace-Links gibt.

Öffnen durch:



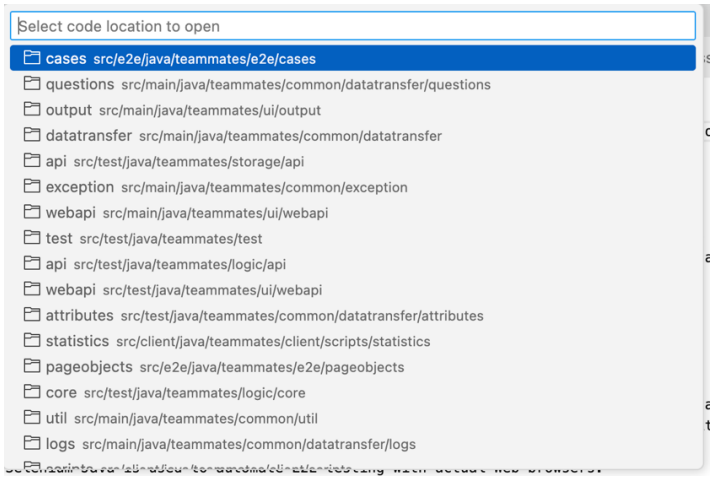
1. Klicken in die Zeile



2. Hovern über der Zeile

3. Status Bar

Öffnet Quick Open:



Directory = wenn 100% der Files ein Trace-Link haben

File

A.3. trace-viz Post Survey

1. Allgemeines Verständnis & Klarheit

1. Die Aufgabenstellung für Sie insgesamt verstädlich?

☐	☐	☐	☐	☐
stimme voll und ganz zu	stimme zu	weder noch	stimme nicht zu	stimme überhaupt nicht zu

2. Hatten Sie zu irgendeinem Zeitpunkt Schwierigkeiten, die Aufgabenstellung zu verstehen?
   ☐ Ja
   ☐ Nein

Wenn ja, welche?

3. Wie klar war für Sie der Zweck der Studie bzw. was von Ihnen erwartet wurde? *(offene Antwort)*

2. Vorgehensweise & Nutzung der Trace-Links

4. Wie sind Sie bei der Bearbeitung der Aufgaben vorgegangen? *(offene Antwort)*

5. Welche Strategien oder Überlegungen haben Ihnen geholfen, die relevanten Stellen zu identifizieren? *(offene Antwort)*

6. Haben Sie während der Bearbeitung Unterschiede zwischen den beiden Varianten (mit und ohne Visualisierung) bemerkt?
   ☐ Ja
   ☐ Nein

Wenn ja, welche Unterschiede waren das?

3. Vergleich der beiden Aufgaben

7. Welche der beiden Aufgaben (mit oder ohne Visualisierung) empfanden Sie als einfacher?
   - ☐ Ohne Visualisierung
   - ☐ Mit Visualisierung
   - ☐ Gleich schwer / kein Unterschied

8. Bitte begründen Sie Ihre Wahl. *(offene Antwort)*

9. Die Visualisierung der Trace-Links bei der zweiten Aufgabe war hilfreich

☐	☐	☐	☐	☐
stimme voll und ganz zu	stimme zu	weder noch	stimme nicht zu	stimme überhaupt nicht zu

10. Die Visualisierung hat Ihr Such- oder Verständnisprozess unterstützt

☐	☐	☐	☐	☐
stimme voll und ganz zu	stimme zu	weder noch	stimme nicht zu	stimme überhaupt nicht zu

4. Wahrgenommener Aufwand & Usability

11. Wie hoch schätzen Sie den Zeitaufwand für jede Aufgabe ein?
    Aufgabe 1: min
    Aufgabe 2: min

12. Hatten Sie das Gefühl, ausreichend Zeit zu haben, um beide Aufgaben zu lösen?
    - ☐ Ja
    - ☐ Nein

    Wenn nein, warum?

13. Gab es Situationen, in denen Sie sich unsicher fühlten, wie Sie weitermachen sollten? *(offene Antwort)*

5. Nutzungserlebnis & Präferenz

14. Die Nutzung der Visualisierung war intuitiv für Sie?

☐	☐	☐	☐	☐
stimme voll und ganz zu	stimme zu	weder noch	stimme nicht zu	stimme überhaupt nicht zu

15. Würden Sie ein solches Visualisierungstool im realen Arbeitskontext verwenden?
    - ☐ Ja
    - ☐ Nein
    - ☐ Unsicher

Bitte erläutern Sie Ihre Antwort.

16. Welche Funktionen oder Darstellungsformen der Visualisierung fanden Sie besonders hilfreich? *(offene Antwort)*

17. Welche Funktionen oder Darstellungsformen empfanden Sie als verwirrend oder unnötig? *(offene Antwort)*

6. Wahrnehmung & Verbesserungsvorschläge

18. Haben Sie das Gefühl, dass die Visualisierung Ihre Effizienz oder Genauigkeit verbessert hat?

☐	☐	☐	☐	☐
stimme voll und ganz zu	stimme zu	weder noch	stimme nicht zu	stimme überhaupt nicht zu

19. Haben Sie Verbesserungsvorschläge für die Visualisierung oder den Ablauf der Studie? *(offene Antwort)*

20. Möchten Sie noch etwas ergänzen, das Ihnen während der Bearbeitung aufgefallen ist? *(offene Antwort)*

21. Sind Sie generell mit Traceability-Links vertraut?

☐	☐	☐	☐	☐
stimme voll und ganz zu	stimme zu	weder noch	stimme nicht zu	stimme überhaupt nicht zu

22. Sind Sie vertraut mit dem Projekt **TeamMates**?
    - ☐ Ja
    - ☐ Nein

23. Wie oft arbeiten Sie in Ihrem Alltag mit ähnlichen Werkzeugen oder Konzepten (z. B. Traceability, Visualisierung, Softwareanalyse)? *(offene Antwort)*

7. Demographie

24. Altersgruppe
    - ☐ Unter 20 Jahre
    - ☐ 20–24 Jahre
    - ☐ 25–29 Jahre
    - ☐ 30–39 Jahre
    - ☐ 40–49 Jahre
    - ☐ 50 Jahre oder älter
    - ☐ Bevorzuge keine Angabe

25. Geschlecht
    - ☐ Weiblich
    - ☐ Männlich
    - ☐ Divers
    - ☐ Bevorzuge keine Angabe

26. Bildungs- / beruflicher Hintergrund
    - ☐ Bachelorstudent:in
    - ☐ Masterstudent:in
    - ☐ Promovierende:r
    - ☐ Wissenschaftliche:r Mitarbeiter:in
    - ☐ Berufstätig in der Industrie
    - ☐ Sonstiges (bitte angeben):

27. Fachrichtung / Studiengang
    (offene Antwort)

28. Selbsteinschätzung der Programmiererfahrung
    (1 = keine Erfahrung, 5 = sehr erfahren)
    - ☐ 1
    - ☐ 2
    - ☐ 3
    - ☐ 4
    - ☐ 5

29. Erfahrung im Bereich Software Engineering
    (1 = keine Erfahrung, 5 = sehr erfahren)
    - ☐ 1
    - ☐ 2
    - ☐ 3
    - ☐ 4
    - ☐ 5

30. Wie lange beschäftigen Sie sich bereits mit Softwareentwicklung oder Softwareanalyse?
    - ☐ Unter 1 Jahr
    - ☐ 1–2 Jahre
    - ☐ 3–5 Jahre
    - ☐ 6–10 Jahre
    - ☐ Mehr als 10 Jahre

Falls Sie an den Ergebnissen der Studie interessiert sind, können Sie optional Ihre E-Mail-Adresse angeben: