REVERSE ENGINEERING MICROSERVICES

REVERSE ENGINEERING MICROSERVICES USING SINGLE SOURCE OF TRUTH FOR ENHANCED INSIGHTS

BY

MUHAMMAD WAQAR UL HASSAN AWAN, BSc

A Report

submitted to the Computing and Software

and the School of Graduate Studies

of McMaster University

in partial fulfilment of the requirements

for the degree of

Master of Engineering

Master of Engineering (2025) McMaster University

(Computing and Software) Hamilton, Ontario, Canada

TITLE:	Reverse Engineering Microservices using Single Source of Truth for Enhanced Insights
AUTHOR:	Muhammad Waqar Ul Hassan Awan M.Eng. Computing and Software, McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Sébastien Mosser
NUMBER OF PAGES:	xii, 74

Lay Abstract

Software maintenance, such as debugging and issue resolution, is a key part of the software development process. However, maintaining an extensive software system with multiple interconnected components is easier said than done. One approach to improving software maintenance is reverse engineering the system for more straightforward analysis and insights. This report presents a comprehensive framework that reverse engineers the static source code into a visualized format by centralizing key information artifacts from the software in one place. The study demonstrates this approach using an actual software application and real-world use cases. This method aids the maintenance process by providing valuable insights and facilitating analysis.

Abstract

Modern software systems have become significantly complex with the growing demand for features, and the need for them to be efficient and reliable has also increased. To manage this complexity, software developers have adopted advanced architectures. However, over the years, as new features are added, software systems tend to become less reliable, requiring regular maintenance. Maintaining such large systems is not a simple task, and a significant amount of resources is needed just to understand and debug even small issues. Among the various solutions to address this problem, reverse engineering appears to be one of the most feasible options, as it helps analyze the problem at hand. Surprisingly, the tools available for reverse engineering large distributed systems are limited, and those that do exist are not very flexible in terms of supporting different technologies or focusing on specific parts of an application. This report presents a framework capable of reverse engineering the static source code of any distributed system using the Unified Data Source (UDS) approach. We will consider real-world scenarios that commonly arise during software maintenance and use a microservices-based application to demonstrate the framework's effectiveness. By reverse engineering specific parts of the application, we aim to validate the practicality and credibility of our approach in real-world applications.

Acknowledgements

I want to express my sincere gratitude to my supervisor, **Dr. Sébastien Mosser**, for his support, guidance, and patience throughout my master's studies. His knowledge and mentorship have been invaluable, and I truly appreciate the time and effort he has put into helping me complete this report. His advice and encouragement made it easier for me to complete this project.

A big thank you to my **family**, especially my mom and dad, for always supporting me in every way possible. Your encouragement, love, and financial support have made this journey easier, and I can't thank you enough for always believing in me.

I also want to thank McMaster University for giving me the opportunity to study in such a great environment. A special thanks to all my professors for their guidance, teaching, and support throughout my master's degree.

Contents

List of Figures

Snippets

Abbreviations

Abbreviations

SST	Single source of truth
UDS	Unified data source
RE	Reverse engineering
SRE	Software reverse engineering
DSL	Domain-specific language
CI	Continuous Integration
CD	Continuous Deployment
VCS	Version Control system
JSON	JavaScript Object Notation

Chapter 1

Introduction

Developing a large software system is a complex and crucial process that requires careful planning and execution. When a stable product is built using a monolithic architecture, where a single codebase handles all business logic, years of development—adding new features and data—can make it highly prone to errors and less resilient. To address this issue, the industry adopted the *"Divide and Conquer"* principle and migrated their products to microservices architecture, where each service represents a separate business logic. However, a major drawback of microservices is the difficulty of maintaining them due to multiple interconnected parts. For example, Monzo, a UK-based digital bank, has implemented a system comprising approximately 2,800 microservices and counting (Monzo Engineering Team, 2024).

Some companies, including Amazon's Prime Video team, have reverted from microservices to monolithic architectures due to challenges in maintaining microservices. As a result, they reduced infrastructure costs by 90% and improved scalability (Anderson, 2023). Maintaining such large systems is just as crucial as building them. One effective approach to understand the internal structure of a system to make it

easier to maintain is **reverse engineering**. Reverse engineering helps analyze complex and legacy systems by leveraging automatic visualization techniques to manage large systems effectively. It also aids maintainers in analyzing source code at different levels of abstraction (Koschke, 2003).

There are no one-size-fits-all tools available in the market that adopt the reverse engineering approach for software maintenance. For example, *Rigi*[1] is a tool for software reverse engineering that visualizes legacy systems. However, Rigi has limited language support, as its built-in parsers primarily support C and COBOL. Users have also reported performance issues when analyzing large systems, particularly with graphs exceeding 500 nodes (Koschke, 2002). Active development of Rigi ceased in 1999, with the last official release in 2003.

We can present a framework that can reverse engineer large distributed systems by performing static analysis on the source code and extracting useful artifacts. This process can be integrated into the CI/CD pipeline to extract real-time information with each release.

The primary objective of this report is to demonstrate such a framework that extracts artifacts from source code and uses the *unified data source (UDS)* approach to maintain up-to-date and credible data. The extracted information is stored as nodes and edges in a graphical database, which is then connected to a visualizer for further analysis and insights based on specific requirements.

This report will answer the following questions:

1. Which key components must be integrated to effectively reverse engineer any microservice architecture based system?

[1] `https://rigi.uvic.ca/Pages/download.html`

2. How can data collected by probes be centralized into a unified, consistent, and real-time source to enhance accuracy and reliability?

3. How can stored data from the source code be transformed into visually intuitive and insightful graphical representations?

Chapter 2 provides background information and key points essential for understanding the overall concept of this project. Chapter 3 discusses the envisioning of the framework and its three key components, which address Question 1. Chapter 4 covers the implementation of probes and their integration with the unified data source, answering Question 2. Chapter 5 validates the scenarios discussed in Chapter 3 by generating graphical information and insights using the data stored in the UDS, addressing Question 3. Lastly, Chapter 6 concludes this report and explores future work that could enhance the framework's usability and practical applications.

The full source code of the probe component implementation is available in the project's GitHub repository[2]. For anyone interested in seeing the entire process, a video demonstrating the framework is available on YouTube[3].

[2] `https://github.com/WaqarAwan376/MEng-Project/releases/tag/v1.0.3`
[3] `https://www.youtube.com/watch?v=jkvvtTBqES8&ab_channel=ACEResearch`

Chapter 2

Background

This chapter provides the necessary background information to understand this project. It begins with a discussion on the software development and its lifecycle, followed by an overview of the software maintenance process after the main development phase. Next, it states a brief summary of the concepts of monolithic and microservices software architectures, which are essential for understanding this report. Finally, it covers the basics of reverse engineering in distributed systems before moving on to the technical strategy and implementation of the framework chapters.

2.1 Software Development and Lifecycle

A large scale software system involves many interconnected components, all of which must adhere to essential software development principles. The goal is not just to write code but to build a system that is reliable, maintainable, scalable, and efficient. Ensuring the system is free of bugs and capable of adapting to future needs is as important as the initial development itself. By following these key paradigms,

developers can create software that meets high standards of quality and performance.

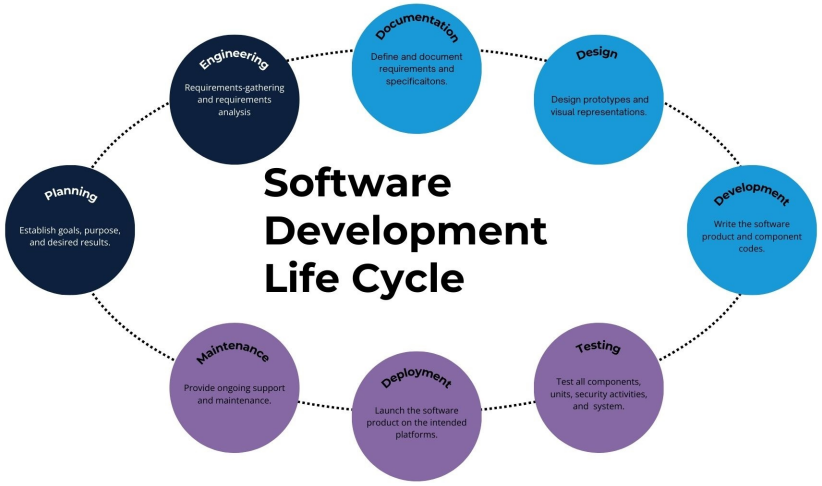

Figure 2.1: Software Development Life cycle
(adapted from Sire (2024))

Figure 2.1 illustrates the software development life cycle. Each phase of the software development lifecycle process adds an important contribution to the overall project. For example, in the planning and engineering phase, the team works closely with stakeholders to determine the functional and non-functional requirements of the project. This collaboration ensures that everyone involved understands the project's goals and technical needs. In the documentation phase, all the information gathered during planning and engineering is carefully recorded. This creates a detailed reference for the team, helping maintain consistency and clarity as the project progresses.

Following the *Software Development Life Cycle (SDLC)* provides several key benefits. It helps in identifying clear goals, ensures all stakeholders are on the same page, and allows for thorough testing at every stage. This structured approach produces high-quality software systems and maintains a smooth and understandable development flow. By sticking to the *SDLC*, teams can reduce risks, avoid confusion, and

create software that meets user expectations.

Once the major development work is complete, the focus shifts to software maintenance. Maintaining a large project becomes a significant task in itself. Updates, bug fixes, and adapting to new requirements or technologies are ongoing responsibilities. Without proper maintenance, even the best-designed systems can become outdated or difficult to use.

Another challenge that arises is understanding the system once it has been completed. Software development often requires years of effort and large amount of resources. As a result, understanding the full complexity of a system after its development can be difficult, especially for teams that were not part of the original project. Proper documentation, clear workflows, and thorough knowledge transfer are critical to addressing this issue. However, even with these measures, there are instances where critical system insights are required to understand the system. This is why reverse engineering is often applied to existing or legacy systems to gain a clear understanding of their structure and functionality.

2.2 Software Maintenance for Large-Scale Systems

Since an already built large-scale software system is quite complex, understanding this system for maintenance purposes requires certain strategies and tools. Moreover, resolving a problem in complex software architectures, such as microservices or service-oriented architectures, often demands significant resources. These architectures consist of numerous interconnected components, and identifying the root

cause of an issue can be challenging. The process may involve extensive debugging, analyzing logs, coordinating between multiple teams, and sometimes even re-evaluating design decisions. This can result in considerable time, effort, and cost being spent to restore functionality and ensure the system operates smoothly. The post-development usability issues in software systems sometimes require significant architectural changes (Folmer et al., 2005). In order to deal with such architectural changes, an understanding of the whole system is required, and if the system is large enough, major resources are spent fixing even minor issues.

Multiple surveys indicate that software maintenance consumes 60% to 80% of the total life cycle costs. Also, the maintenance costs are largely due to enhancement (often 75-80%), rather than corrections (Canfora and Cimitile, 2001). To address these challenges, there is a growing need for tools that can assist in resolving bugs and reducing maintenance overhead. Such tools should be capable of reverse engineering software systems to provide a clear and comprehensive view of the architecture. By highlighting the key components and their interactions, these tools make it easier for developers to understand the system, identify issues, and perform necessary tasks efficiently. This not only simplifies debugging but also enhances the overall maintainability of the software, ensuring smoother operation and reduced downtime.

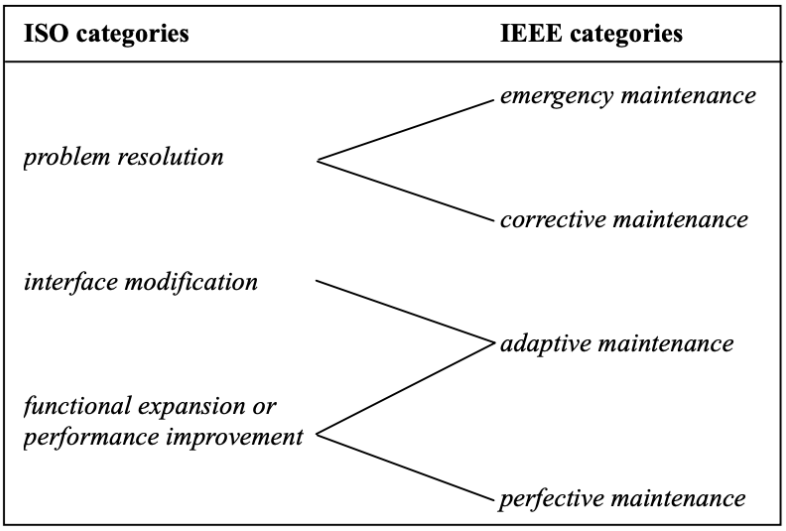

Figure 2.2: ISO and IEEE maintenance categories
(adapted from Canfora and Cimitile (2001))

2.3 Software Architectures

Software architecture is the fundamental structure of a software system. Software architectures *"represents the design decisions related to overall system structure and behavior. Architecture helps stakeholders understand and analyze how the system will achieve essential qualities such as modifiability, availability, and security"* (Software Engineering Institute, Carnegie Mellon University, [n. d.]).

Each architecture has its pros and cons. There are different types of software architectures adopted or sometimes introduced in order to solve certain issues. The most important ones that are necessary to be understood for this report are monolithic and microservices architectures.

2.3.1 Monolithic Architecture

Monolithic architecture is a traditional software development design paradigm where an application is built as a single, unified unit. All the components of the system are tightly coupled and dependent on each other. The monolithic architecture is simple, easier to design, develop and test and usually have easier deployment process as compared to some other architectures.

2.3.2 Microservice Architecture

Microservice architecture is a type of distributed system architecture and a software design in which a system is built as a collection of small, independent, and loosely coupled services. Each service is designed to keep in mind the Single-responsibility principle and hence has a specific function, operates independently and communicates with other services typically using HTTP or messaging queues. Distributed system architectures are easier to scale, flexible, autonomous and more resilient than other architectures, especially monolithic.

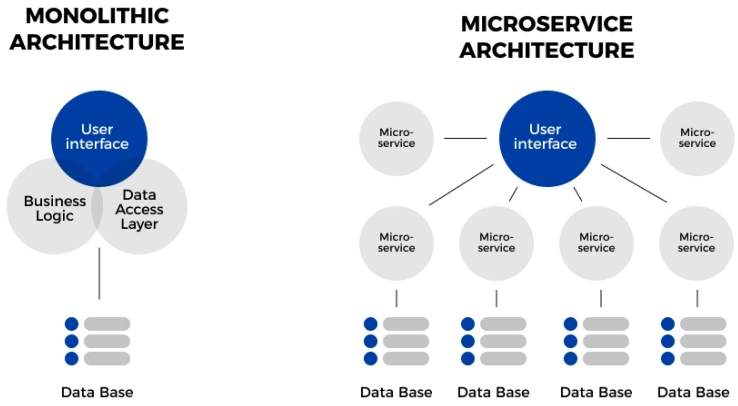

Figure 2.3: Different between Monolithic & Microservices Architecture (adapted from Harris (2024))

2.3.3 Spring Petclinic

Spring petclinic[1] is a Java Spring framework based application that demonstrates best practices for building Java Spring applications. It was originally built using the monolithic architecture[2], and then later on, the application system was split into independent services to demonstrate the best practices for Java Spring-based microservices[3], Spring Boot, and Spring Cloud. The microservices-based Petclinic project is a good candidate for the case study since it is actively maintained and it's widely used in the developer community for learning microservices best practices.

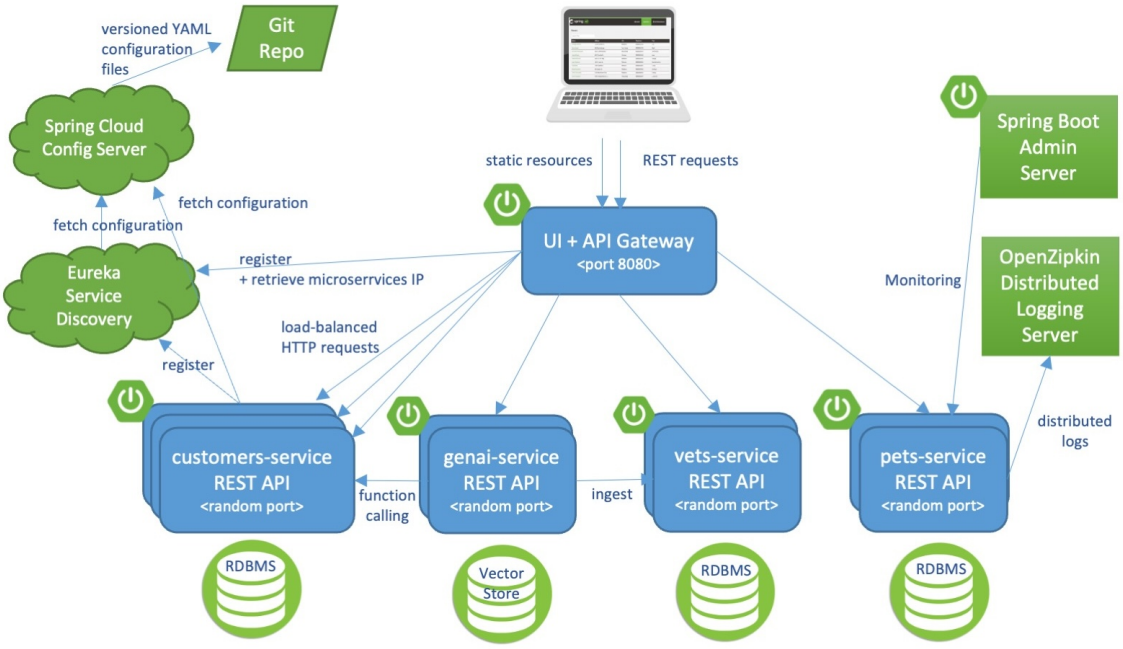

Figure 2.4: Architecture diagram of the Spring Petclinic Microservices (adapted from Spring Team (2025))

[1] https://spring-petclinic.github.io/
[2] https://github.com/spring-projects/spring-petclinic
[3] https://github.com/spring-petclinic/spring-petclinic-microservices

2.4 Maintenance and Reverse Engineering(RE)

Ever since software systems started becoming more and more complex, many of the larger systems that were based on monolithic architectures started migrating their systems to microservices-based architectures. Figure 2.5 illustrates the microservices architecture of Uber in mid-2018. Because microservices and distributed systems are so complex, so is their maintenance. Moreover, there are other maintenance-related issues, like monitoring and collecting logs from independent microservices deployed in containers, and since microservices interact asynchronously, debugging failures is more complicated (Waseem et al., 2021). Maintenance of such systems requires thorough understanding, and since understanding such an extensive system is a complex process that cannot be accomplished in a single day, some other technique is needed to speed up the process. That is where reverse engineering plays a vital role. *"The goal of reverse engineering is to reveal the logic, features, and functionalities embedded within the software"*(Digital.ai, 2023). Different techniques are employed to reverse engineer the system and reveal its logic and functionalities.

2.4.1 RE Approaches

There are three essential approaches for carrying out software reverse engineering. *Observation-based analysis* which involves studying the exchange of information within the software to infer its functionality and behaviour. *Disassembly* approach uses a disassembler to interpret and analyze the program's raw machine code. *Decompilation* approach uses a decompiler to attempt a reconstruction of the program's source code in a high-level language, starting from machine code or bytecode (Oladipo et al., 2012).

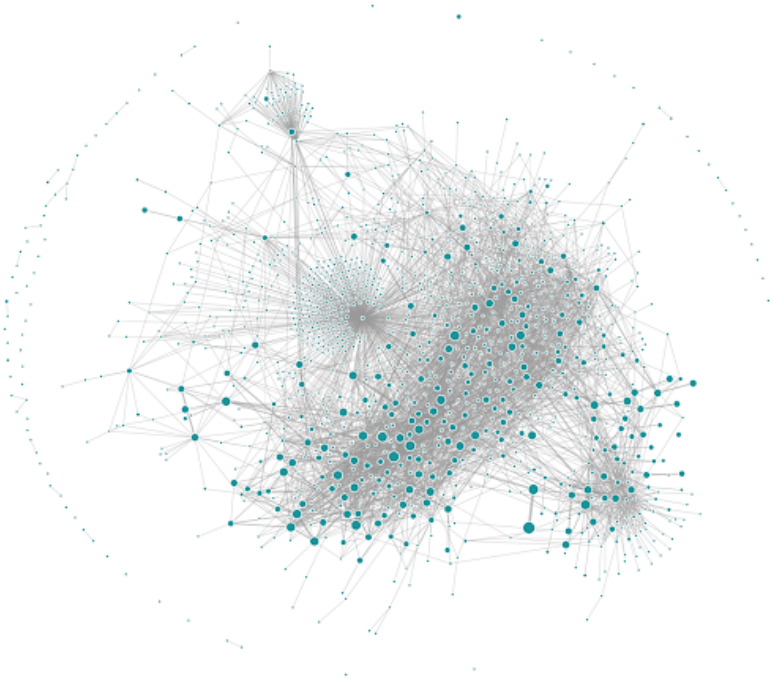

Figure 2.5: Uber's microservice architecture circa mid-2018 from Jaeger (adapted from Gluck (2020))

2.4.2 Reverse Engineering: Challenges and Insights

Reverse engineering comes with several challenges. Rene R. Klosch discusses some of the challenges of reverse engineering (Klösch, 1996). The paper mentions that legacy systems have source code that is poorly structured, which makes them challenging to analyze and understand. Large and complex systems require significant effort and expertise to be deconstructed. Moreover, maintenance engineers often lack sufficient knowledge about the original application domain, which is crucial for reverse engineering. The documentation, on the other hand, is often outdated or nonexistent, resulting in a lack of explicit information about the system's functionality and design.

2.4.3 RE for Distributed Systems Maintenance

Reverse engineering plays an essential role in maintaining and understanding large distributed systems. The approach used in the process depends on the requirement and overall specification of the project. In his thesis, Dan Calin Cosma discusses the role of reverse engineering in distributed systems maintenance (Cosma, 2010). He mentions that reverse engineering helps analyze the structure of distributed software systems, which often have complex dependencies due to their distributed nature. He also mentions that it can help identify functional units within the distributed systems. Furthermore, the thesis emphasizes that reverse engineering is not just about understanding the current system but also enabling its evolution by refactoring the current system without disrupting existing flows.

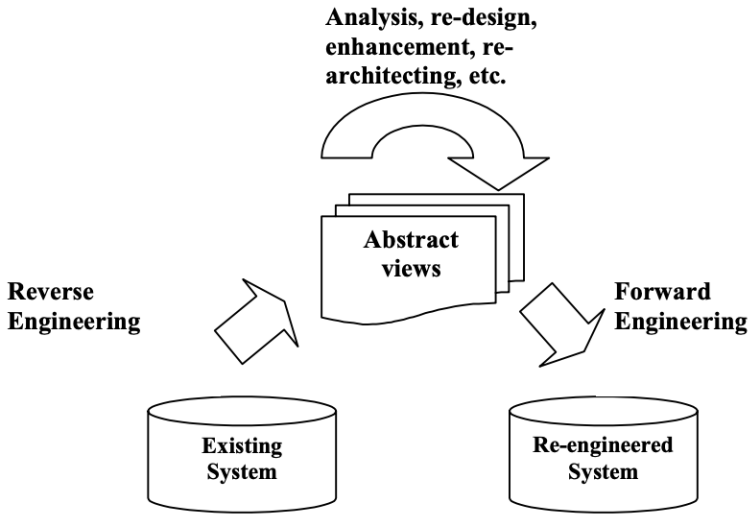

Figure 2.6: Reverse engineering and re-engineering
(adapted from Canfora and Cimitile (2001))

Chapter 3

Approach: UDS-Driven Reverse Engineering Strategy

In this chapter, we will explore the necessity of a framework for software analysis and its role in addressing critical challenges in modern software systems. We will outline the overall vision of the framework, highlighting its unique features and how it stands apart from existing solutions. Finally, we will discuss the technical strategy employed to guarantee the framework's functionality, we will delve into the use cases of probes designed for the framework validation, ensuring it can effectively address real-world challenges.

3.1 Vision

Keeping in view the discussion in the previous chapter, there is a need for an approach in which the software system could be analyzed, undergo processing, and produce useful information that can be used to provide enhanced insights about the software

system. There are several tools available for software analysis, each with its strengths and weaknesses.

3.1.1 Designing the Framework

The vision is to work on a framework that stands out due to key differences. The most notable feature we are looking for is platform and technology independence. This means that the tool should not be tied to a specific technology. This flexibility will allow it to be written in any programming language, depending on the requirements, enabling data extraction from various software systems. Moreover, the tool should be able to run static code analysis on microservices and standalone services as well. Another distinctive feature is the integration of the extracted data with the unified data source. It should be capable of handling diverse types of data, generalizing it, and storing it in a graph-based database. This approach will eliminate the immediate need for a separate visualization tool. In cases where appropriate data representation is already available in the database, users can perform analyses directly without additional tools. Finally, our tool should not rely on a single visualizer for data representation. Since the data should be generalized, any compatible visualizer can be used with minimal adjustments to meet specific needs and requirements. This flexibility will enhance usability and ensure that the tool can adapt to diverse scenarios efficiently. In summary, the vision is that our tool should offer unmatched flexibility in data extraction, storage, and visualization, setting it apart from existing solutions in the field of software analysis.

In this report, we will discuss, implement, and validate a framework by extracting static information from a project. In the future, this can be implemented/integrated with the project deployment pipeline and provide dynamic information from the projects.

3.1.2 System Context and Approach

The test project used in this report is a Java Spring framework-based project called **Petclinic**, which is mentioned in subsection 2.3.3. It is important to note that all the data we extract from the system is specific to this project setting. The artifacts that extract the data from the projects are supposed to be written for the subject systems depending on their requirements, settings and technology.

Moreover, our approach will mainly focus on the unified data source (UDS) technique. *"Unified Data refers to the integration and consolidation of data from various sources into a single, cohesive framework. This approach allows organizations to streamline their data management processes, ensuring that all data is accessible and usable across different departments and applications. By unifying data, businesses can eliminate silos, reduce redundancy, and enhance the overall quality of their data analytics efforts"* (Statistics-Easily, 2025). This means that consistent, up-to-date, and valid data will be available using the UDS technique.

3.1.3 Validation Roadmap: Key Scenarios

In order to demonstrate the working of the framework, we will propose eight real-world scenarios and use cases. These use cases will be based on actual scenarios that can be faced in the maintenance phase of the software development life cycle. All of

these scenarios will fall under the umbrella of software maintenance.

1. **Most Recent Contributor:** A bug is found in a method, the most recent author who updated the method would have the context of the recent changes that they have made and can help diagnose and resolve the issue faster. We have to find the author who has made the most recent changes to the method.

   Rationale: Promotes accountability among team members by making contribution history transparent. Increases team efficiency and issue resolving by involving right people for the job.

2. **List of Contributors:** A class and its method are responsible for a feature in the application. A list of people who worked on the method is required for documentation purposes. Find the list of all the contributors of a particular method.

   Rationale: Facilitates and improves communication between team members by identifying relevant stakeholders. Improves resource allocation, project planning and decision making for development tasks.

3. **Top Contributor:** A bug is found in the method. The top contributor of the method will most likely be the subject matter expert. Identify the top contributor of the method.

   Rationale: This will help the organization increase its efficiency and productivity.

4. **File Contributors:** The project manager needs to contact contributors of a file for clarifications on specific changes or potential bugs. Identify the list of all the developers who worked on a particular file.

   Rationale: Assigning issues to contributors who are familiar with the relevant

files improves resolution speed. Knowing who contributed to a file ensures accountability and encourages higher-quality contributions. Identifies files relying on single contributor which can help in workload distribution.

5. **Author Relation:** Identify quantitative measure of collaboration between two developers in a development team. Higher strength shows frequent joint contributions.

   Rationale: Improve team dynamics and encourage better interaction in areas with weaker team collaboration. Developers with high collaboration strength are likely to solve issues in shared files effectively. Management can use the data to evaluate employees. Developers with higher collaboration strengths with multiple individuals show employee value.

6. **Endpoints:** An API endpoint is a URL that acts as the point of contact between an API client and an API server(The Postman Team, 2023). Extract all the REST API endpoints information to generate accurate and up-to-date documentation and perform analysis.

   Rationale: Helps locating the faulty file and class, and helps developers quickly identify the method handling the request and resolve the issue. Teams can use extracted endpoint data, providing clients with API documentation. Facilitates communication between backend and frontend teams by providing endpoint insights.

7. **Bean Data:** A bean is an object that is instantiated, assembled, and managed by a Spring IoC(inversion of control) container(Spring Framework Documentation, 2025). The project manager needs to get an overview of all registered beans and

their relationships from the java spring based application. Extract bean data within the project to reveal the dependencies between components.

Rationale: Quickly locates missing or misconfigured beans, reducing the development downtime. Detects potential security risks and stability issues.

8. **Dependencies List:** Some conflicting or outdated dependencies are causing issues in the Maven-based project. Extract dependencies within each service to help track the versions of libraries and frameworks used across microservices.

   Rationale: Helps to identify, update or remove unused dependencies optimizing the performance and efficiency of the services. Make it easy to identify those dependencies that are impacting the security of the services. Help keep all the dependencies up-to-date and easy to maintain.

The upcoming section will discuss the technical strategy for addressing the scenarios mentioned above.

3.2 Structural Elements of the Framework

This section provides a detailed discussion of the main components of the framework. It also includes an analysis of the probes, their use cases, and their benefits. After that, it discusses what UDS is and, finally, the visualizer. Figure 3.1 shows the conceptual working of the framework's three main components: Probes, UDS, and Visualizer.

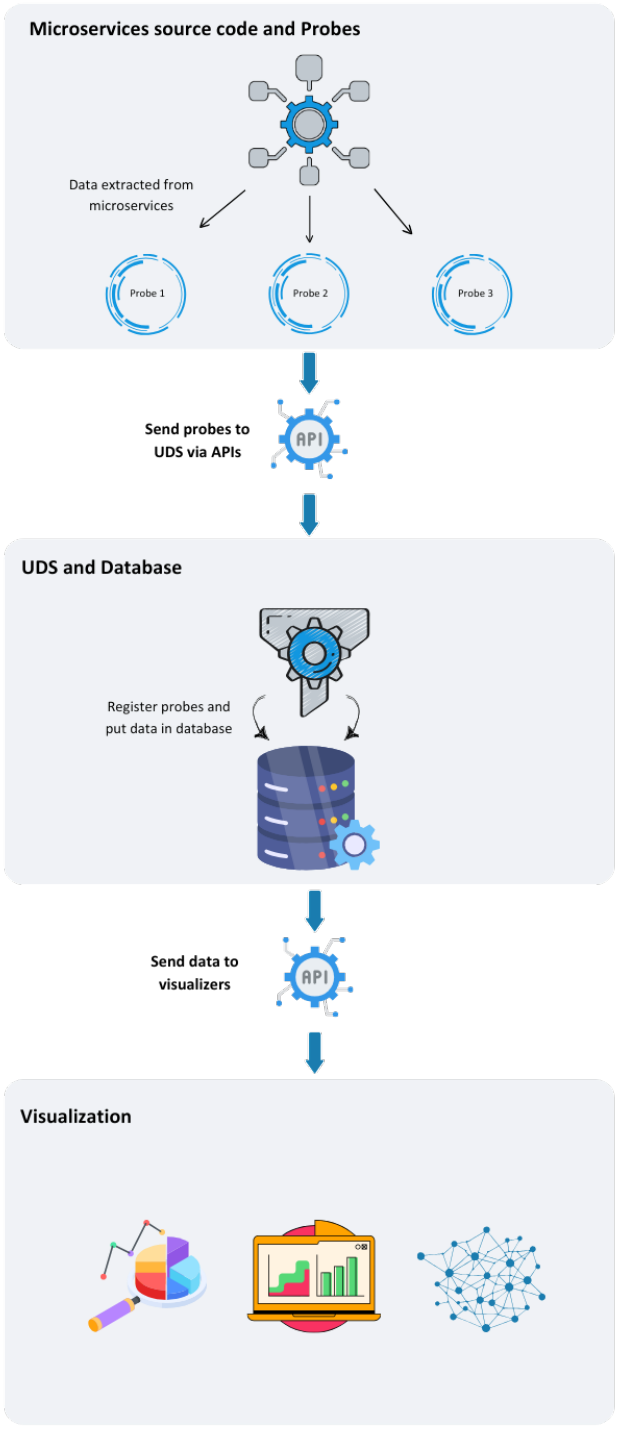

Figure 3.1: Working of Probes, UDS and Visualizer

3.2.1 Probes: Extractors

The first component of the framework consists of *probes*. Probes represent distinct informational artifacts that are extracted from software systems to provide insights and actionable data. In the context of this report, we have several use cases to extract specific pieces of information, details of which are given in the sections below. These probes will help demonstrate the usage of the framework and serve as the foundational elements for gathering critical data points that enable analysis and decision-making.

Looking forward, this concept can be expanded to more complex and targeted informational needs. By refining the scope and nature of the probes, we can tailor them according to our needs and capture more refined data. This flexibility ensures that as the systems grow or change, the framework remains relevant and capable of producing deeper, more impactful information.

3.2.2 Unified Data Source (UDS)

When data is being extracted from diverse sources, there are high chances of it becoming inconsistent and stale. In section 3.1, the visualizer component is mentioned, which runs the desired analysis on the data and provides visual updates. So, if analysis is to be done on the input data, it needs to be accessible, credible, reliable, and consistent.

Maintaining data quality is crucial for accurate insights and decision-making. Unified sources ensure that the input data remains clean and updated, regardless of its origin. Properly validated and processed data serves as the foundation for meaningful analysis.

A scalable and extensible approach exists for software analysis and visualization

in which data from diverse sources is integrated into a unified source. A *unified data source* provides a single access point for querying diverse data, eliminating the need to manage multiple disconnected sources (Müller et al., 2018).

There are multiple UDS tools available, but the tool used in this framework is the one developed by our colleague, Stepan Bryantsev, a member of the McMaster University McSCert lab[1]. This tool called *single source of truth (SST)*, consumes data from the probes, creates relationship mappings, and stores it in graphical form using the Neo4J database[2]. It is working as a data warehouse that takes all the information from diverse sources and process it. A detailed discussion on the working of the single source of truth can be found in chapter 4, and the validation of its output will be discussed in chapter 5.

3.2.3 Integration and Output

The probes will extract the required data from the microservices and feed them to the SST for further processing. The SST can be integrated with appropriate tool for further analysis and visualization.

3.2.4 Data Management

The output of the SST will be stored in a separate database to maintain and preserve historical records. It will help ensure that the data is structured, organized, and easily retrievable from a centralized repository. By consolidating the data into a single repository, it not only simplifies access but also certifies a consistent format.

[1] `https://www.mcscert.ca/`
[2] `https://neo4j.com/`

Storing the data in a database enhances its integrity by eliminating inconsistencies and reducing complexities, such as duplicate entries. This structure allows analysts or analysis tools to focus on extracting meaningful insights without the additional work of cleaning or reorganizing raw data.

Additionally, a dedicated database supports scalability, meaning it can accommodate larger datasets as the system evolves. It also offers improved data management capabilities, such as access controls and backup solutions.

3.2.5 Visualizer

After software analysis is completed with the help of probes and SST, we can use some visualizer tools in order to produce useful information gathered from the software system. There is a diverse range of software visualization and analysis tools available in the market, with each of them having their strengths and weaknesses. Sarita Bassil and Rudolf K. Keller cover more than 40 tools for software visualization, with each having their advantages (Bassil and Keller, 2001). For example, Rigi is used for understanding legacy systems, GraphViz is popular for graph-based visualizations, and daVinci offers dynamic graph layouts.

Following are some of the important tools offered by AWS and Azure for visualization.

- **Amazon QuickSight**: A Business Intelligence (BI) service that enables users to visualize data through dashboards, interactive graphs, and analytics. It connects to various data sources, including software systems, databases, and AWS services, and performs real-time data analysis. Read more about this from (Amazon Web Services - QuickSight, 2025).

- **Amazon X-Ray**: A service for analyzing and debugging applications. It collects data about requests to your applications, including errors and performance bottlenecks. Visualizes the application architecture in a service map for easy identification of issues. Read more about this from (Amazon Web Services - X-Ray, 2025).

- **Azure Monitor**: Azure monitor collects, analyzes, and visualizes telemetry data from Azure and on-premise environments to monitor application performance and detect issues. Key features are logs and metrics collection for in-depth analysis, provides alerts and insights to troubleshoot issues in real-time (Microsoft, 2024).

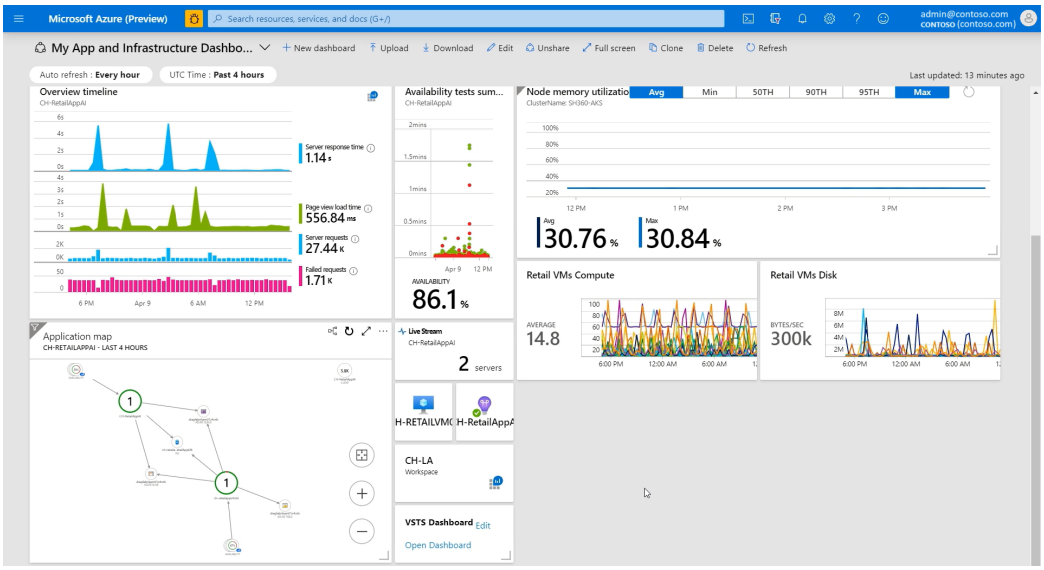

Figure 3.2: Dashboard of Azure monitor
(adapted from Microsoft - Azure Monitor Best Practices (2024))

In our report, after integrating the probes and SST, the processed data from SST will be used to visualize the output generated by the probes.

3.3 Technical Validation Strategy

In this section, we will discuss the technical strategy that we will adopt to tackle the scenarios and use cases mentioned in subsection 3.1.3.

3.3.1 Effective Methods to Identify Contributors, Top Contributors, and Recent Contributors

The scenarios starting from 1 to 5, related to the authors/contributors, are merged in this section to avoid repetition since all of them require a similar approach. There are several ways to find the contributors list, author relation, top contributors, and most recent contributors:

- Reviewing communication records for recent discussions or code reviews about the method. However, this process is slow and often leads to inconsistent results.
- Checking pull request histories on GitHub, where reviewers and contributors are listed. For repositories with many pull requests, filtering relevant changes can be overwhelming.
- Analyzing the repository's commit history directly from platforms like GitHub or Bitbucket. Commit messages often do not clearly indicate which methods were changed, making this process extremely time-consuming.
- Using the GitHub API to programmatically extract and analyze contributors' data related to specific files. However, API usage may be restricted by rate limits, especially for large repositories.

- Using CI/CD tools like Jenkins to monitor and report code changes and associate them with contributors. This can slow down CI/CD pipelines and requires initial setup and ongoing maintenance.
- Checking project documentation and looking for change logs that might include information about recent changes and contributions. However, change logs are not always updated regularly, which can lead to inaccurate results.

One of the most effective ways to achieve this is by using a version control system. In almost every project, some form of version control is used, as it allows tracking the author of the most recent changes to specific lines of code. Each project typically includes a version control file. In our test project, a `.git` directory is available, which we can parse to retrieve the required information. To achieve this, we will develop a few probes to perform static code analysis, parse the `.git` directory, and extract the necessary data to feed it into the SST. The implementation of these probes will be discussed in later chapters.

Figure 3.3 shows the process of extracting Git data. The Python script runs a Git command that returns data for each line of code. It then parses this information and extracts the required details.

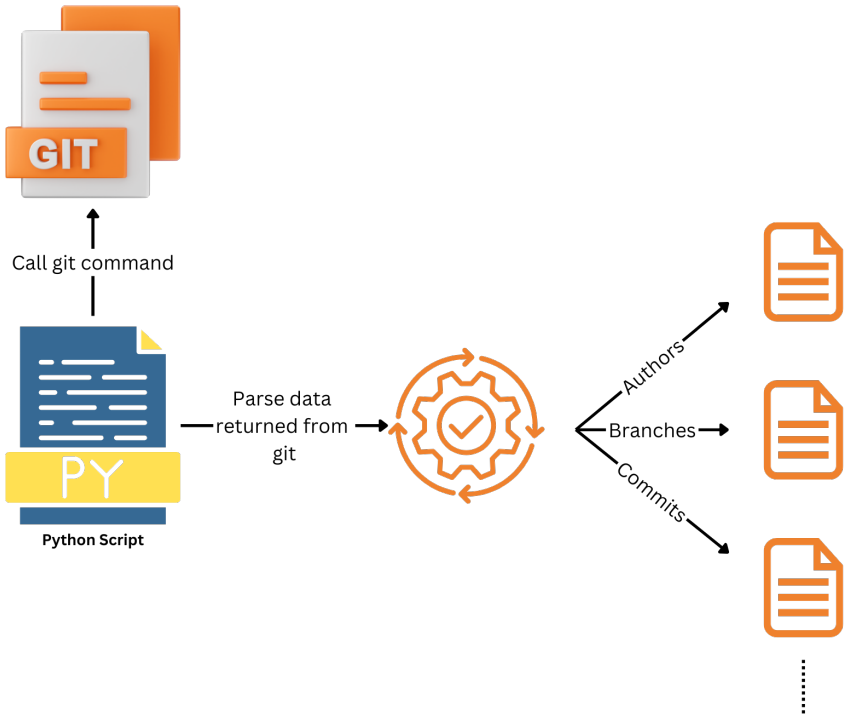

Figure 3.3: Parsing Git Files

3.3.2 Microservices Endpoints

For the 6th scenario, we have to determine the endpoints. *REST API* endpoints for microservices can be identified in several ways from the source code. Our goal is to establish a general method for obtaining this information from your project, as real-world microservice architectures do not always use frameworks like Java Spring, as seen in the Petclinic test project referenced in this report. Below are some approaches to discovering endpoints:

- Integrating Swagger[3] or OpenAPI[4] to auto-generate documentation can be effective but requires accurate configuration for all endpoints. Misconfiguration

[3] `https://swagger.io/`
[4] `https://swagger.io/resources/articles/documenting-apis-with-swagger/`

can lead to incorrect results.

- Enabling logging of all incoming requests to capture and log accessed endpoints at runtime can also work. However, this approach introduces performance overhead and may miss endpoints that are rarely accessed or currently unused.

- Spring Boot Actuator provides an endpoint mapping system, but it only works for services based on the Spring framework.

The approach we will use involves static code analysis. We will write a probe to search for `@RequestMapping`, `@GetMapping`, `@PostMapping`, and similar annotations in the source code to identify endpoints. This probe can be configured to match the target service setup and extract REST API endpoints, which will then be stored in a unified data source.

3.3.3 Beans And Dependencies

For the 7th and 8th scenarios, We have to extract beans and dependencies from the source code. There are several ways to identify the beans and dependencies in the Spring framework project under discussion, allowing us to visualize this data for better documentation and understanding:

- Spring Boot Actuator provides a `/actuator/beans` endpoint that lists all beans in the application context, along with their dependencies and initialization details. However, this can cause performance issues in large applications with many beans.

- The Spring *ApplicationContext* can be accessed programmatically to retrieve

a list of all beans and their dependencies (Peterlić, 2024). However, this approach requires adding custom code for bean inspection, which can increase the codebase size and potentially clutter production logs.

The approach we will use involves static code analysis. We will develop a probe to search for annotations like `@Bean`, `@Component`, `@Service`, `@Repository`, and other dependency-related annotations. Additionally, we will parse the `POM.xml` files of the services to identify the dependencies associated with each service.

3.3.4 SST and Visualizer Integration

For the 9th and 10th scenarios, integration of probes with SST and SST with visualizer is required. After extracting the necessary information, we need to demonstrate the complete capabilities of our framework by applying it to the technical scenarios we have designed. To achieve this, it is essential to integrate the extracted data with the SST with the help of REST API endpoints provided by the SST tool. Once probes and SST start exchanging data, our next task is to connect at least one visualizer tool with the SST. For that, we will connect a visualizer tool with the SST's database to obtain real-time data and information directly from the database.

The following chapter will discuss details about how the probes are integrated with SST and visualizer.

Chapter 4

Implementation

In the previous chapters, we discussed the vision and technical strategy we will employ to demonstrate the framework's complete working. This chapter will discuss the critical implementation points, especially those related to probes and the single source of truth. We will discuss what approach to use for extracting data through probes, how probes are integrated with the SST, how the runner script will run all the probes and send the data to the SST, how data is stored in the SST, and how we can use it to integrate with a visualizer for further analysis.

4.1 Inside the SST Tool: Registration, Data Integration, and Storage

We have already discussed at length how data extracted from the probes can be plugged into the SST tool, which works as a unified data source. In this section, we will explore the SST tool, its purpose, and how it operates. We will begin by

discussing how to run this tool. After that, we will examine the types of data that its endpoints accept, ensuring that we implement our probes correctly according to the REST API structure.

Later subsections will focus on how data is collected from probes, sent to the SST tool, and stored. This step-by-step breakdown will help us understand the complete workflow, from data input to storage.

4.1.1 Running the SST Tool

The SST tool depends on two main services:

- **SST Backend Service** – This service sends and receives probe data.
- **Neo4j Database** - This database stores the collected data in a graph-based format, making it easier to visualize and analyze relationships between data points.

To set up SST, we first need to clone the source code from the GitHub repository[1]. After cloning, we require a *docker-compose* file to initialize, which is available on Docker Hub[2]. By using this setup, we ensure that all components of the SST tool are properly configured and ready for use. Once both of the above steps are completed, run the services using `docker-compose up`. We can check the backend services running by sending a get request to `/api/health-check/` endpoint.

[1] `https://github.com/ace-design/uds`
[2] `https://hub.docker.com/r/acedesign/sst`

4.1.2 Probe Registration

All probes must be registered first to ensure that the data remains consistent. The registration process ensures that the SST system can correctly store, process, and relate the incoming data. To register a probe, the following three elements are required:

- Probe Name – The name of the probe that is being registered.
- Nodes – The data elements that will be provided to the SST system.
- Edges – The relationships between the nodes.

Each node represents an object that holds specific data. It is important that each node is unique and contains a value that serves as its unique identifier. This identifier is crucial because SST merges nodes based on this value. If a node with the same identifier already exists in the system, the original one will be retained, preventing duplication.

Edges define how nodes are connected, establishing relationships between them. These relationships help structure the data in a meaningful way, allowing for efficient querying and retrieval within the system.

Listing 4.1: Sample JSON for Probe Registration

```
{
    "name": "Authors_Relation",
    "types": [
        {
            "name": "Author",
            "fields": [
                {
                    "name": "email",
                    "type": "string",
                    "unique": true
                }
```

```
            ],
            "mergeRules": [
                {
                    "fieldName": "email"
                }
            ]
        }
    ],
    "relations": [
        {
            "name": "Collaborated",
            "from": {
                "typeName": "Author"
            },
            "to": {
                "typeName": "Author"
            }
        }
    ]
}
```

The above sample JSON shows the data required to register a probe. `types` is an array of JSON objects, where each object in this array represents a node type. Each node has a `name`, an array of `fields`, and `mergeRules`. The `fields` represent the object attributes for the node, while `mergeRules` define the value on the basis of which two nodes will be merged.

`relations` is another array of JSON objects, where each object represents a relationship between two nodes. Each relation has a `name`, a `from` field, and a `to` field. The `from` and `to` fields define the node types between which the relationship should be established. This structure ensures that data is correctly linked within the SST system, allowing for efficient merging and retrieval of information.

This probe represents that all the nodes from this probe will be of type *Author*, and each author node will have a relation between them of type *Collaborated*. Each

node will be connected on the basis of `mergeRules` key, which in this case is *email*.

Once the structure of the probe is defined, it must be registered using the following REST API provided by the SST tool. Use post request and send the JSON data to the SST. If the probe is registered, it will return a response status code of 201, and a status of *success* will be shown in the response object.

4.1.3 Probe Integration

Once the probe has been registered, the next process is to extract the data from the source code for integration with SST. Make sure the data should be in the same structure as the structure that is used to register the probe.

Listing 4.2: JSON for Author Relation Probe. This represents the same scenario as shown in snippet 4.1

```
{
    "probeName": "Authors_Relation",
    "nodes": [
        {
            "type": "Author",
            "name": "James␣Rey",
            "email": "james@gmail.com"
        },
        {
            "type": "Author",
            "name": "Allen",
            "email": "allen@gmail.com"
        }
    ],
    "edges": [
        {
            "relationName": "Collaborated",
            "from": {
                "nodeType": "Author",
                "propertyName": "email",
                "propertyValue": "james@gmail.com"
            },
```

```
            "to": {
                "nodeType": "Author",
                "propertyName": "email",
                "propertyValue": "allen@gmail.com"
            }
        }
    ]
}
```

4.2 Implementing Probes for Data Collection and Analysis

In the previous section, we discussed how the single source of truth works and how probes are registered and integrated with it. In this section, we will focus on the implementation of probes to demonstrate the functionality of the framework and validate the scenarios outlined in subsection 3.1.3.

We will cover only the key aspects of the implementation to provide a clear understanding of the process to the reader. For those interested in exploring the complete implementation in detail, the full source code is available at the Project's GitHub Repository[3].

We have implemented classes for each node and edge to keep the code consistent and ensure that all objects follow the same structure. We have also created classes to define relationships between nodes, ensuring that the data remains consistent for similar objects.

[3] https://github.com/WaqarAwan376/MEng-Project/releases/tag/v1.0.3

4.2.1 Authors and Version Control

As mentioned in section 3.3, we have merged the discussion and implementation of probes related to authors/contributors. We have discussed that an effective way to retrieve data related to authors and their contributions is by using a *version control system*. Version control systems inherently track all changes made to a codebase and store this information within the `.git` directory, which is typically hidden. This directory contains the complete history of modifications, including details about contributors, timestamps, and commit messages, making it a reliable source for extracting author-related data.

We have several Python[4] scripts to extract information from a git directory. These scripts analyze data such as the relationship between two authors based on the number of files they have collaborated on, identifying all authors who worked on a method, determining the top contributor among them, and listing the authors of each file.

Each script relies on the `.git` directory and uses the *Javalang library*[5] to parse Java source code. This library helps navigate Java code structure using *Abstract Syntax Tree (AST)*, making it easier to extract relevant information. After parsing the Java source code, we extract author details of each line of the source code using the git blame command[6].

Once the author's information is extracted, it is stored in a JSON file and sent to SST for further processing and analysis.

For the sake of demonstrating the probe structure, we can take method contribution as an example to show the JSON structure for probes related to authors.

[4] `https://www.python.org/`
[5] `https://pypi.org/project/javalang/`
[6] `https://git-scm.com/docs/git-blame`

Appendix A.3 provides a sample JSON representation for the author method contribution probe. This example illustrates that the `Owner.java` file contains the `Owner` class, which includes the `getPetsInternal()` method. Rest of the probes related to authors can be explored in the project repository.

4.2.2 Microservices REST API endpoints

In section 3.3, we discussed a strategy to extract REST API endpoints from *Java Spring Framework* source code. The approach involves searching for relevant annotations and decorators in the source code and using them to identify endpoints. We apply this logic to extract the required information.

In the implemented Python script, we first extract Java files from all services. Then, we use regular expressions to check if they contain annotations such as `@RestController`, `@RequestMapping`, `@GetMapping`, `@PostMapping`, and similar decorators. If a file includes these annotations, we extract the full route information and create class and endpoint nodes. This allows us to represent how a particular class maps to an endpoint through relational edges.

Appendix A.4 shows a sample JSON for endpoint mapping. The JSON shows that file `VetResource.java` contains `VetResource` class. This class maps `/vets` endpoint.

4.2.3 Java Beans Extraction

Using a similar approach to extracting REST API endpoints, we can also extract Java beans from the source code. Java beans are created using specific annotations placed at the top of a class. These annotations indicate that the class is a Java bean, which will be managed by the Java Spring Framework.

In the Python script we have implemented, we search for `@Component`, `@Service`, `@Repository`, `@Controller`, and `@Configuration` annotations in the source code to identify beans.

Once the beans are found, we also extract their methods by searching for the `@Bean` annotation. In this script, we use *javalang* to parse methods when identifying bean methods and Python regular expressions to find classes in the source code. After determining the beans and their methods, we create their nodes using Python classes in our code. This allows us to establish relational edges between them.

Appendix A.5 shows a sample JSON for bean classes and methods mapping. The `AIBeanConfiguration.java` file contains `AIBeanConfiguration` class. This class has a bean method `loadBalancedWebClientBuilder()`.

4.2.4 Java Dependencies Extraction

Java services have a *Project Object Model (POM)* file that contains information about the Maven project[7] and how it is built. It is an XML file that includes a list of all project dependencies. We can use this file to extract dependencies and their versions.

In the Python script we have implemented, we use a lightweight Python XML parser[8] for this purpose. We parse the XML file, extract the dependencies from each service, and create nodes and edges based on this information. When a dependency is found in the POM file, a node is made using the dependency class in our code. These nodes are then linked to each other using edges to represent relationships between dependencies.

Appendix A.6 shows a sample JSON for dependencies graphs. The POM file

[7] `https://maven.apache.org/`

[8] `https://docs.python.org/3/library/xml.etree.elementtree.html`

in `spring-petclinic-genai-service` contains `spring-ai-bom` dependency. The properties key of their relation includes further information about the dependency, showing its version, scope, and type.

4.2.5 Runner script and How it works

We have a Python runner script that is responsible for running each probe script. To manage these probes, we use a `probes_list.py` script file, which contains a list of Python dictionaries. Each dictionary represents a probe and includes three key elements: *runner_command*, *probe_file*, and *arguments*. The runner script iterates through this list, executing each probe and sending a `POST` request to the SST server to update it with the latest extracted data. The `probes_list.py` script file can be seen in appendix section A.1.

One of the key advantages of this approach is that probes do not have to be written exclusively in Python. As long as a probe is placed in the `probes` folder and its corresponding runner command and probe file are specified in `probes_list.py` file with correct arguments, it can be executed in the same way as other probes. This flexibility allows for the integration of scripts written in different programming languages.

The runner script ensures that the data sent to the SST server is always up-to-date, consistent, and valid. It takes the source directory of the target source code as a command-line argument, allowing users to specify the location of the project they want to analyze. Below is an example of how to run the script:

```
python runner.py
−−SOURCE_DIR "/Projects/spring-petclinic-microservices"
−−DIR_NAME "spring-petclinic-microservices"
```

This method ensures an efficient and automated way to extract and update data while maintaining consistency across different probes.

Appendix A.2 shows the implementation of the runner script. The script parses the command-line argument passed to it and runs the probe from the probes list, producing outputs. It is important to note that the `runner.py` script only sends the probe's nodes and edges data. The registration of the probes must be done manually for now. The script uses the requests[9] Python dependency to send requests to the SST server.

4.3 Data Visualization and Insights

Now that the probe and SST tools are integrated and the required data has been successfully extracted from the source, we can proceed to use various visualization tools to analyze this data—completing the reverse engineering process by uncovering statistical insights and valuable information aligned with our objectives. There are numerous tools and services available for data visualization. We have already discussed some key visualization tools in subsection 3.2.5. In this section, we provide a brief summary of the visualization tools used in this report.

[9] `https://pypi.org/project/requests/`

4.3.1 Neo4j Desktop

Neo4j provides its own application, *Neo4j Desktop*[10], to visualize data stored in a Neo4j database. Users can connect to both local and remote Neo4j servers. The application includes several tools for visualizing different types of data.

One key tool is *Neo4j Browser*[11], a web-based query tool for running Cypher queries. Another is *Neo4j Bloom*[12], which allows users to explore graph data interactively.

4.3.2 Tableau

Tableau is a leading data visualization tool used for data analysis and business intelligence (Biswal, 2023). It is known for its ease of use, compatibility with diverse data sources, and ability to create interactive dashboards that enhance data exploration and decision-making[13]. One of the key reasons for choosing Tableau is its flexibility in handling various types of data while offering intuitive visualizations.

Since our SST tool stores data in a Neo4j database, Tableau provides a way to integrate with Neo4j, allowing real-time data capture and analysis. Additionally, it supports visualizing relationships between data points, aligning with Neo4j's graph-based structure.

[10] `https://neo4j.com/download`
[11] `https://neo4j.com/docs/browser-manual/current/`
[12] `https://neo4j.com/product/bloom/`
[13] `https://www.tableau.com/`

4.3.3 Tool Selection and Justification

In this report, we will use Tableau and Neo4j Desktop to demonstrate the functionality of the probes and SST framework and validate our scenarios. Neo4j Desktop will be used for managing and visualizing graph-based data, while Tableau, which provides a way to connect directly with the Neo4j database, will assist in creating insightful visual representations. Using these tools, we aim to illustrate the framework's workings and ensure proper scenario validation.

Chapter 5

Scenario Validations

In this chapter, we examine the output generated by the probes that we implemented. We will also go through each validation scenario as outlined in subsection 3.1.3. While this chapter primarily focuses on scenario validations, demonstrating the framework's functionality in realistic scenarios, it also touches on aspects of verification. Specifically, we assess whether the framework as a whole behaves as expected when applied to real-world software systems. Although there is no strict set of formal requirements being verified, the successful execution of the defined scenarios serves as an implicit check of the framework's correctness and completeness. After reviewing the results, we will discuss the limitations and challenges of the framework, highlighting areas that could be improved.

5.1 Evaluating the Framework in Practical Applications

Since the framework we are developing is intended for real-world projects, this validation aims to demonstrate its full functionality using real-world scenarios. The objective is to show how the framework operates in realistic projects, making sure relevant data is extracted by probes, ensuring its effectiveness in handling real software systems. As discussed earlier, each scenario represents common use cases that can arise during project maintenance. Since reverse engineering is mainly used to break down complex software into smaller, manageable components, it becomes a valuable tool in the maintenance and evolution of software projects. By dissecting an existing system, it helps developers understand its structure, dependencies, and functionality, making future modifications and improvements more manageable.

It is important to note that evaluating the framework does not solely rely on the current probes' output, as the overall results can vary depending on the probes used for each project. Instead, this validation focuses on whether the framework as a whole functions as expected, regardless of the specific probes implemented. To establish its working, the real-world scenarios defined in this report must demonstrate that the framework successfully processes and analyzes software systems as intended. In short, the goal is to provide a structured and functional approach to reverse engineering, ensuring that it can be effectively applied in practical scenarios.

5.2 Validation Results

In this section, we will go through each validation scenario discussed in subsection 3.1.3. We will review the output, but not all output figures are included here. For a complete set of images, refer to Appendix B.

5.2.1 Most Recent Contributor

It shows the contributor who made the most recent changes to the method. To achieve this, we extracted methods and authors as nodes.

The expected output for this probe is a method node containing detailed information about the method. This node should be linked to author nodes, representing all contributors who have modified the method. Among these connections, a relation should indicate the most recent contributor, determined by the latest modification date.

Instead of displaying all methods and their authors (which can be viewed in the Appendix B), we will take one method as an example and present its results.

We can use Neo4j Cypher queries[1] to filter and retrieve the required data.

Listing 5.1: Most recent contributor cypher query

```
1 MATCH (n)-[r]-(m)
2 WHERE n.signature = "org.springframework.samples.petclinic.customers.web.
    OwnerResource.createOwner(org.springframework.samples.petclinic.
    customers.web.OwnerRequest)"
3 RETURN n, r, m
```

This will extract information only about the method based on our probe settings and the output JSON. We can further interact with the nodes and edges in the Neo4j

[1] https://neo4j.com/docs/cypher-manual/current/queries/

browser to explore additional relationships.

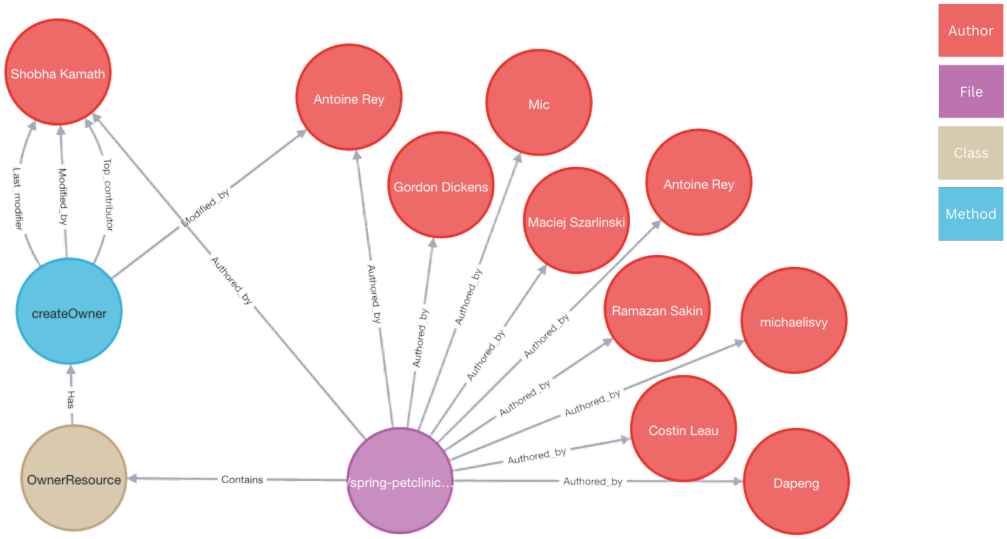

Figure 5.1: Neo4j Browser showing the relation between nodes

Figure 5.1 illustrates the node relationships in the Neo4j Browser visualizer. We have used the query mentioned above to extract the data and extend class and file relations. Additional data and details can be included in each relationship edge. For example, the `Last_modifier` relationship displays the last modification date and time.

5.2.2 List of Contributors

It shows all the contributors of the methods. The expected output for this probe is a set of author nodes and methods.

Figure 5.1 illustrates two authors who modified the `createOwner` method. The author nodes contain information about the authors, such as their names and email addresses, while the method nodes store details about the method, including a unique identifier. In the current probes, we uniquely identify a method by combining the

package name of the Java file, the class name, and the method signature. Authors are identified using their email addresses, as multiple contributors may have the same name, but email addresses are always unique.

5.2.3 Top Contributor

It identifies the top contributor to the method. Instead of creating a separate node for the top contributor, we establish a relationship edge from the method node to the author node, labeled as `Top_contributor`. This relationship includes properties that indicate the total lines added by the top contributor, along with the specific code line numbers of their contributions.

Figure 5.1 illustrates two authors who modified the method `createOwner`. Among them, the author *Shobha Kamath* has a `Top_contributor` relationship edge attached, indicating that they are the top contributors to this method.

5.2.4 File Contributors

It displays the list of contributors for each file. To achieve this, we extracted files and authors as nodes. Like the top contributor approach, we represent file authorship by establishing a direct relationship between files and authors. An edge from a file node to an author node, labeled as `Authored_by`, indicates that that particular contributor authored the file.

Figure 5.1 illustrates all the current authors of the file.

5.2.5 Author Relation

It represents the joint contributions between two authors, measured by the number of file collaborations. The strength of the connection between the two authors indicates the number of shared contributions.

We can show this by connecting the names of two authors to the `Author_Relation` node. This node contains data such as the strength of their collaboration and a list of files they have worked on together.

Since this analysis is more statistical, we can visualize it using *Tableau*. Figure 5.1 illustrates the relationship strength between authors. The highlighted section shows an example of a quantitative connection between two contributors. Further analysis can be performed based on specific requirements, such as sorting by increasing contribution strength or integrating the data into an organization's dashboard to visualize team collaboration.

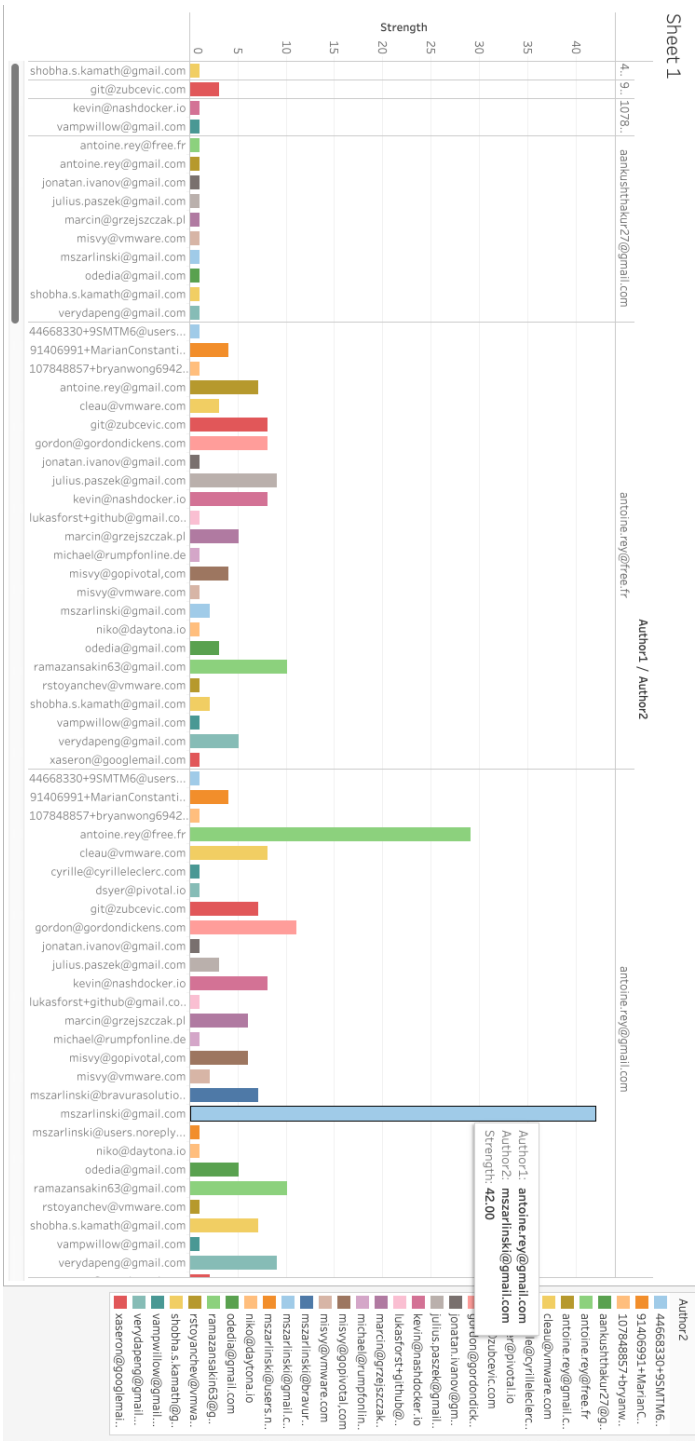

Figure 5.2: Tableau tool showing authors' relation strength

5.2.6 REST API Endpoints

It displays all the REST API endpoints in the project. Methods and endpoints were extracted as nodes using a Cypher query.

Listing 5.2: PetResource class endpoints cypher query

```
MATCH (a)-[r]-(b)
WHERE type(r)="Maps"
AND a.name="PetResource"
RETURN a, r, b
```

The expected output should include class and endpoint nodes. Each class should be connected to its corresponding endpoint nodes using a `Maps` relationship, as a class maps REST API endpoints.

Figure 5.3 illustrates a class node, `PetResource`, mapping four endpoints. The file node shown in the figure was not extracted from the initial query. Instead, it was obtained using the Neo4j Browser node relationship extractor after executing the query.

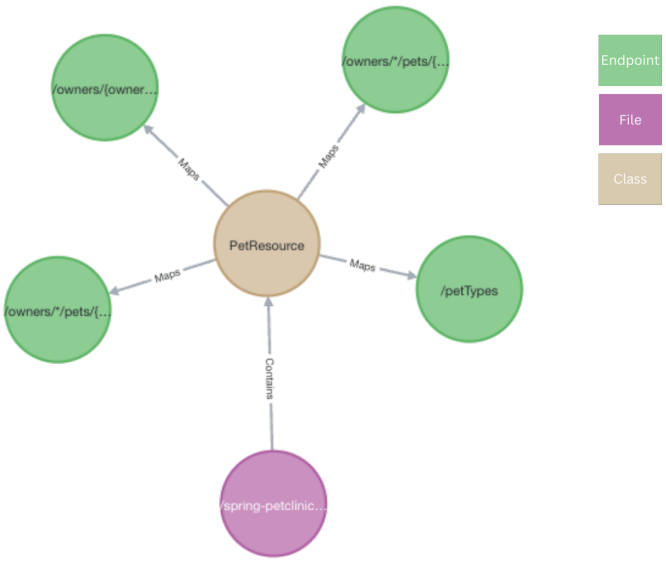

Figure 5.3: Neo4j Browser showing the relation between class and REST API endpoints

5.2.7 Java Beans

It identifies Java Spring bean classes and methods. To achieve this, we extract files, classes, and methods. The expected output includes a file node that has a `has_bean_class` relationship with a class identified as a candidate bean class. This bean class, in turn, has a `has_bean_method` relationship with methods that are considered candidate bean methods.

5.2.8 Dependencies List

It displays all the Maven project dependencies in the project. To achieve this, we locate the *Project Object Model (POM)* file and parse its dependencies.

The expected output consists of a POM file node and multiple dependency nodes. Each dependency contains metadata such as version and scope. Dependencies are

identified using their `artifact_id` and `group_id`.

The POM file node is connected to the dependency nodes through a `depends_on` relationship.

5.3 Challenges and Limitations

Even though the presented framework indeed serves the purpose of reverse engineering software systems and can be used in large codebases and distributed systems, it comes with certain challenges and limitations. While these challenges can be addressed, they require effort and improvements. Below are some of the key challenges and limitations of this framework:

- Currently, the probes can only perform static code analysis. This means the probes must be designed to extract data from static code, which can be challenging if the source code is not written in languages that follow strict coding principles. For example, Java Spring Framework follows *object-oriented programming (OOP)* and *aspect-oriented programming (AOP)* principles, whereas Python does not enforce such strict structures.

- The probes require both domain and code knowledge to be written correctly. If someone is not familiar with the programming language used in the source code, they may struggle to write effective probes. While tools like *SonarQube* can assist in performing static code analysis and extracting data, writing probes manually can be complex without prior knowledge of the codebase.

- A visualizer plays a key role in the reverse engineering process, enabling the user to view insightful information. However, it also comes with challenges. If an external visualizer is not necessary, Neo4j can be used instead, but it requires knowledge of Cypher queries to extract useful information. Additionally, most high-quality visualizers are not free. Learning to use a visualizer becomes an extra step when working with this framework.

Chapter 6

Conclusion and future work

In this chapter, we will present the conclusion of the project and provide an overall summary of the report. Finally, we will outline future directions and the potential roadmap for this project.

6.1 Summary

In conclusion, this report has presented a framework for effectively reverse engineering a software system. The approach involves collecting useful information artifacts using probes, integrating the SST server with the collected data through the UDS approach, and using visualizers to view the extracted information. This process provides valuable insights and analysis, helping in the maintenance of software systems.

Our report began by providing essential background information to help readers understand the purpose and importance of the proposed framework, particularly in the maintenance phase of the software development lifecycle. We highlighted why such a framework is needed and how it can improve software system maintenance.

In Chapter 3, we discussed the overall goal of the framework, detailing its core components and defining validation scenarios to assess its effectiveness. Chapter 4 focused on the implementation process. We started by cloning the Petclinic test project from its official GitHub repository for analysis. Next, we developed probe scripts to extract relevant information based on our predefined validation scenarios.

Following this, we integrated the SST tool as our Unified Data Source (UDS) by setting up its server using Docker and following its documentation. We then connected our probes to the SST tool, registering the structure of probe nodes and edges. Finally, in Chapter 5, we used Neo4j Visualizer and Tableau to present the extracted data, transforming static source code into a visual format for better insights and analysis.

6.2 Future work

The framework demonstrated in this report is its first version, and further testing and additional use cases are required, along with extensive validation. In this section, we will discuss the future roadmap for this project and how it can be further tested through practical applications to be considered *"ready for use"*.

Currently, in the project, the probes can only extract data from static source code. This can be improved by adding probes that capture dynamic data from the software system, such as analyzing communication between services through messaging queues and collecting real-time information on REST API traffic. This could be achieved by probes that continuously monitor logs and transfer data to the SST server for further processing.

The probes currently in use extract data through scripts. Each probe script must

be written by developers with domain or project knowledge. Various methods can be used to extract data from source code, including external tools that scan and analyze it. These external tools can reduce the dependency on custom probes that require domain expertise to be written. Since data extraction from external tools is already well-tested, it might be more reliable, efficient, and applicable across a wider range of use cases and programming languages. Similarly, machine learning processing can be integrated to analyze the extracted data, and the results from this analysis could be used to create separate probes.

The framework can be tested by integrating it into a Continuous Integration (CI) process. This ensures that any code changes are immediately processed by the probes and visualized using SST. This way, real-time analysis and insights can be produced, which will help developers monitor and assess the impact of changes on the source code.

Lastly, the framework can be tested on an enterprise-level application with a large number of probes. This will help evaluate the data handling capabilities of the SST tool and ensure that it does not face performance or reliability issues. Many reverse engineering tools struggle with these aspects when dealing with large datasets, so this testing will be crucial for validating the tool's efficiency and stability.

Appendix A

Scripts and Outputs

A.1 Probes List

```
from utils.constants import OUTPUT_FOLDER

probes_scripts = [
    {
        "runner_command": "python3 -m probes.author_method_contribution",
        "probe_file": "author_method_contribution",
        "arguments": [
            "--PROBE_NAME", "Method_Contributor",
            "--OUTPUT", f"{OUTPUT_FOLDER}{'author_tracking.json'}"
        ]
    },
    {
        "runner_command": "python3 -m probes.authors_files_and_relations",
        "probe_file": "authors_files_and_relations",
        "arguments": [
            "--PROBE_NAME_1", "Authors_Relation",
            "--OUTPUT_FILE_1", f"{OUTPUT_FOLDER}{'author_relation.json'}",
            "--PROBE_NAME_2", "File_Contributors",
            "--OUTPUT_FILE_2", f"{OUTPUT_FOLDER}{'file_contributors.json'}",
        ]
    },
    {
        "runner_command": "python3 -m probes.java_beans",
```

```
        "probe_file": "java_beans",
        "arguments": [
            "--PROBE_NAME", "Beans",
            "--OUTPUT", f"{OUTPUT_FOLDER}{'beans.json'}"
        ]
    },
    {
        "runner_command": "python3 -m probes.services_dependencies",
        "probe_file": "services_dependencies",
        "arguments": [
            "--PROBE_NAME", "Dependencies",
            "--OUTPUT", f"{OUTPUT_FOLDER}{'dependencies.json'}"
        ]
    },
    {
        "runner_command": "python3 -m probes.services_endpoints",
        "probe_file": "services_endpoints",
        "arguments": [
            "--PROBE_NAME", "Endpoints",
            "--OUTPUT", f"{OUTPUT_FOLDER}{'endpoints.json'}"
        ]
    },
]
```

A.2 Runner Script

```
import subprocess
import os
from dotenv import load_dotenv
from probes_list import probes_scripts
from utils.constants import OUTPUT_FOLDER
from utils.helper import get_passed_arguments
import json
import requests

load_dotenv()
SST_API_URL = os.getenv("SST_API_URL")
project_root = os.path.dirname(os.path.abspath(__file__))

if __name__ == '__main__':
```

```
    args = get_passed_arguments("--SOURCE_DIR")
    original_directory = os.getcwd()
    probes_directory = "./probes/"

    for script in probes_scripts:
        subprocess.run(
            f"{script['runner_command']} --INPUT_DIR \"{args.SOURCE_DIR}\" --OUTPUT \
                {OUTPUT_FOLDER}{script['output_file']}", shell=True, cwd=project_root)

    for filename in os.listdir(OUTPUT_FOLDER):
        if filename.endswith(".json"):
            file_path = os.path.join(OUTPUT_FOLDER, filename)

            try:
                with open(file_path, "r", encoding="utf-8") as json_file:
                    json_data = json.load(json_file)

                headers = {"Content-Type": "application/json"}
                response = requests.post(
                    f"{SST_API_URL}/api/upload-graph", json=json_data, headers=headers)

                print(f"Sent {filename}: {response.status_code}")

            except Exception as e:
                print(f"Error processing {filename}: {e}")
```

A.3 Author Method Contribution

```
{
  "probeName": "Methods_Contributions",
  "nodes": [
    {
      "type": "Method",
      "name": "getPetsInternal",
      "signature": "org.springframework.samples.petclinic.customers.model.
          Owner.getPetsInternal()"
    },
```

```
    {
      "type": "File",
      "path": "/spring-petclinic-customers-service/src/main/java/org/
         springframework/samples/petclinic/customers/model/Owner.java"
    },
    {
      "type": "Class",
      "name": "Owner",
      "full_name": "/spring-petclinic-customers-service/src/main/java/org/
         springframework/samples/petclinic/customers/model/Owner.java:Owner
         "
    }
  ],
    "edges": [
    {
      "relationName": "Contains",
      "from": {
        "nodeType": "File",
        "propertyName": "path",
        "propertyValue": "/spring-petclinic-customers-service/src/main/java/
           org/springframework/samples/petclinic/customers/model/Owner.java
           "
      },
      "to": {
        "nodeType": "Class",
        "propertyName": "full_name",
        "propertyValue": "/spring-petclinic-customers-service/src/main/java/
           org/springframework/samples/petclinic/customers/model/Owner.java
           :Owner"
      }
    },
    {
      "relationName": "Has",
      "from": {
        "nodeType": "Class",
        "propertyName": "full_name",
        "propertyValue": "/spring-petclinic-customers-service/src/main/java/
           org/springframework/samples/petclinic/customers/model/Owner.java
           :Owner"
      },
      "to": {
        "nodeType": "Method",
        "propertyName": "signature",
```

```
        "propertyValue": "org.springframework.samples.petclinic.customers.
           model.Owner.getPetsInternal()"
      }
    }
  ]
}
```

A.4 Microservices Endpoints

```
{
  "probeName": "Endpoints",
  "nodes": [
    {
      "type": "File",
      "path": "/spring-petclinic-vets-service/src/main/java/org/
         springframework/samples/petclinic/vets/web/VetResource.java"
    },
    {
      "type": "Class",
      "name": "VetResource",
      "full_name": "/spring-petclinic-vets-service/src/main/java/org/
         springframework/samples/petclinic/vets/web/VetResource.java:
         VetResource"
    },
    {
      "type": "Endpoint",
      "full_method_id": "GET␣-␣/vets",
      "http_method": "GET",
      "route": "/vets"
    }
  ],
    "edges": [
    {
      "relationName": "Contains",
      "from": {
        "nodeType": "File",
        "propertyName": "path",
        "propertyValue": "/spring-petclinic-vets-service/src/main/java/org/
           springframework/samples/petclinic/vets/web/VetResource.java"
      },
```

```
      "to": {
        "nodeType": "Class",
        "propertyName": "full_name",
        "propertyValue": "/spring-petclinic-vets-service/src/main/java/org/
           springframework/samples/petclinic/vets/web/VetResource.java:
           VetResource"
      }
    },
    {
      "relationName": "Maps",
      "from": {
        "nodeType": "Class",
        "propertyName": "full_name",
        "propertyValue": "/spring-petclinic-vets-service/src/main/java/org/
           springframework/samples/petclinic/vets/web/VetResource.java:
           VetResource"
      },
      "to": {
        "nodeType": "Endpoint",
        "propertyName": "full_method_id",
        "propertyValue": "GET␣-␣/vets"
      }
    }
  ]
}
```


A.5 Bean Classes and Methods


```
{
  "probeName": "Beans",
  "nodes": [
    {
      "type": "File",
      "path": "/spring-petclinic-genai-service/src/main/java/org/
         springframework/samples/petclinic/genai/AIBeanConfiguration.java"
    },
    {
      "type": "Class",
      "name": "AIBeanConfiguration",
```

```
      "full_name": "/spring-petclinic-genai-service/src/main/java/org/
         springframework/samples/petclinic/genai/AIBeanConfiguration.java:
         AIBeanConfiguration"
    },
    {
      "type": "Method",
      "name": "loadBalancedWebClientBuilder",
      "signature": "org.springframework.samples.petclinic.api.
         ApiGatewayApplication.loadBalancedWebClientBuilder()"
    }
  ],
    "edges": [
    {
      "relationName": "Has_bean_class",
      "from": {
        "nodeType": "File",
        "propertyName": "path",
        "propertyValue": "/spring-petclinic-genai-service/src/main/java/org/
           springframework/samples/petclinic/genai/AIBeanConfiguration.java
           "
      },
      "to": {
        "nodeType": "Class",
        "propertyName": "full_name",
        "propertyValue": "/spring-petclinic-genai-service/src/main/java/org/
           springframework/samples/petclinic/genai/AIBeanConfiguration.java
           :AIBeanConfiguration"
      }
    },
    {
      "relationName": "Has_bean_method",
      "from": {
        "nodeType": "Class",
        "propertyName": "full_name",
        "propertyValue": "/spring-petclinic-api-gateway/src/main/java/org/
           springframework/samples/petclinic/api/ApiGatewayApplication.java
           :ApiGatewayApplication"
      },
      "to": {
        "nodeType": "Method",
        "propertyName": "signature",
        "propertyValue": "org.springframework.samples.petclinic.api.
           ApiGatewayApplication.loadBalancedWebClientBuilder()"
```

```
      }
    }
  ]
}
```

A.6 Dependencies


```
{
  "probeName": "Dependencies",
  "nodes": [
    {
      "type": "File",
      "path": "/spring-petclinic-genai-service/pom.xml"
    },
    {
      "type": "Dependency",
      "group_id": "org.springframework.ai",
      "artifact_id": "spring-ai-bom",
      "combined_name": "org.springframework.ai:spring-ai-bom"
    }
  ],
  "edges": [
    {
      "relationName": "Depends_on",
      "from": {
        "nodeType": "File",
        "propertyName": "path",
        "propertyValue": "/spring-petclinic-genai-service/pom.xml"
      },
      "to": {
        "nodeType": "Dependency",
        "propertyName": "combined_name",
        "propertyValue": "org.springframework.ai:spring-ai-bom"
      },
      "properties": {
        "version": "1.0.0-M4",
        "scope": "import",
        "type": "pom"
      }
    }
```

```
34   ]
35 }
```

Appendix B

Neo4j Browser Visualization Images

The images related to scenario validations can be found on the project's GitHub Repository[1]. The following list includes the brief details of each image.

1. **method_modifiers.png[2]:** Displaying the most recent and top contributors among all contributors of the method. The `Top_contributor` and `Last_modifier` relations contain additional data, such as the number of lines updated by the top contributor and the date of the last modification, respectively.

```
MATCH (a)-[r]->(b)
WHERE type(r) IN ["Last_modifier", "Modified_by", "Top_contributor"]
AND a.signature="org.springframework.samples.petclinic.customers.web.
    PetResource.processCreationForm(org.springframework.samples.
    petclinic.customers.web.PetRequest,int)"
AND NOT a:Type AND NOT b:Type
RETURN a, r, b
```

[1] https://github.com/WaqarAwan376/MEng-Project/tree/master/Report/Visualization_Images

[2] https://github.com/WaqarAwan376/MEng-Project/blob/master/Report/Visualization_Images/method_modifiers.png

2. **method_authors.png**[3]**:** Displays all methods along with their authors and modifiers. Each method also includes a relation indicating the last modifier and the top contributor. In this example, type nodes are excluded for clarity. The total number of methods, authors, and relations is also shown.

```
MATCH (a)-[r]->(b)
WHERE type(r) IN ["Last_modifier","Modified_by","Top_contributor"]
AND NOT a:Type AND NOT b:Type
RETURN a, r, b
```

3. **files_authors.png**[4]**:** Displays the authors of each file. The relationship is represented by the `Authored_by` edge. The total number of files, authors, and relations is also shown. Clicking on any node or edge reveals its information.

```
MATCH (a)-[r]->(b)
WHERE type(r)="Authored_by"
AND NOT a:Type AND NOT b:Type
RETURN a, r, b
```

4. **file_contributors.png**[5]**:** Displays the contributors of a specific file. In this example, the file `VisitRepository.java` has three authors.

```
MATCH (a)-[r]->(b)
WHERE type(r)="Authored_by"
AND a.path="/spring-petclinic-visits-service/src/main/java/org/
    springframework/samples/petclinic/visits/model/VisitRepository.java"
AND NOT a:Type AND NOT b:Type
RETURN a, r, b
```

[3]https://github.com/WaqarAwan376/MEng-Project/blob/master/Report/Visualization_Images/method_authors.png

[4]https://github.com/WaqarAwan376/MEng-Project/blob/master/Report/Visualization_Images/files_authors.png

[5]https://github.com/WaqarAwan376/MEng-Project/blob/master/Report/Visualization_Images/file_contributors.png

5. **author_relation.png[6]:** Displays the joint collaboration between two authors. The two author nodes are connected to an `Author_Relation` node through a collaborated relationship. The `Author_Relation` node contains a summary of the collaboration, including its strength and a list of files the two authors have worked on together.

```
MATCH (a)-[r]->(b)
WHERE type(r)="Contributed"
AND b.combined_emails="antoine.rey@gmail.com:marcin@grzejszczak.pl"
AND NOT a:Type AND NOT b:Type
RETURN a, r, b
```

6. **endpoints_map.png[7]:** Displays the classes that map REST API endpoints. The green nodes represent endpoints, each containing additional information, such as the request type.

```
MATCH (a)-[r]->(b)
WHERE type(r)="Maps"
AND NOT a:Type AND NOT b:Type
RETURN a, r, b
```

7. **classes_files_ep.png[8]:** Shows that files "contain" classes, and the classes "map" REST API endpoints.

```
MATCH (a)-[r1:Contains]->(b)-[r2:Maps]->(c)
WHERE NOT a:Type AND NOT b:Type
RETURN a, r1, b, r2, c
```

[6] https://github.com/WaqarAwan376/MEng-Project/blob/master/Report/Visualization_Images/author_relation.png

[7] https://github.com/WaqarAwan376/MEng-Project/blob/master/Report/Visualization_Images/endpoints_map.png

[8] https://github.com/WaqarAwan376/MEng-Project/blob/master/Report/Visualization_Images/classes_files_ep.png

8. **ownerResource_endpoints.png**[9]**:** Shows that the `ownerResource.java` file "contains" the OwnerResource class, which "maps" four REST API endpoints.

```
MATCH (a)-[r1:Contains]->(b)-[r2:Maps]->(c)
WHERE NOT a:Type AND NOT b:Type
AND a.path="/spring-petclinic-customers-service/src/main/java/org/
    springframework/samples/petclinic/customers/web/OwnerResource.java"
RETURN a, r1, b, r2, c
```

9. **bean_classes_methods.png**[10]**:** Displays the files that contain bean classes and the bean classes that contain bean methods.

```
MATCH (a)-[r]->(b)
WHERE type(r) IN ["Has_bean_class","Has_bean_method"]
AND NOT a:Type AND NOT b:Type
RETURN a, r, b
```

10. **metricConfig_beans.png**[11]**:** Displays the `MetricConfig.java` file, which contains a class configured as a bean class. This class has two bean methods: `timedAspect` and `metricsCommonTags`. Note: The "Contains" relation between the file and class, and the "Has" relation between the class and methods, come from another relation.

```
MATCH (a)-[r]->(b)
WHERE type(r) IN ["Has_bean_class", "Has_bean_method"]
AND NOT a:Type AND NOT b:Type
AND b.full_name="/spring-petclinic-customers-service/src/main/java/org/
    springframework/samples/petclinic/customers/config/MetricConfig.java
    :MetricConfig"
OPTIONAL MATCH (b)-[r2]-(c)
RETURN a, r, b, r2, c
```

[9] https://github.com/WaqarAwan376/MEng-Project/blob/master/Report/Visualization_Images/ownerResource_endpoints.png

[10] https://github.com/WaqarAwan376/MEng-Project/blob/master/Report/Visualization_Images/bean_classes_methods.png

[11] https://github.com/WaqarAwan376/MEng-Project/blob/master/Report/Visualization_Images/metricConfig_beans.png

11. **pom_dependencies.png**[12]**:** Displays all dependencies related to POM files. Dependencies are uniquely identified using a combination of *artifactId* and *groupId*. Each dependency node includes additional details, such as version and scope.

```
MATCH (a)-[r]->(b)
WHERE type(r)="Depends_on"
AND NOT a:Type AND NOT b:Type
RETURN a, r, b
```

12. **dependency_data.png**[13]**:** Displays that four POM files depend on the dependency `org.hsqldb:hsqldb`. On the right side, the relationship between the POM file and the dependency also includes the scope as part of the metadata.

```
MATCH (a)-[r]->(b)
WHERE type(r)="Depends_on"
AND NOT a:Type AND NOT b:Type
AND b.combined_name="org.hsqldb:hsqldb"
RETURN a, r, b
```

[12] https://github.com/WaqarAwan376/MEng-Project/blob/master/Report/Visualization_Images/pom_dependencies.png

[13] https://github.com/WaqarAwan376/MEng-Project/blob/master/Report/Visualization_Images/dependency_data.png

Bibliography

Amazon Web Services - QuickSight. 2025. *Amazon QuickSight Documentation*. https://docs.aws.amazon.com/quicksight/latest/user/welcome.html Accessed: 2025-01-16.

Amazon Web Services - X-Ray. 2025. *AWS X-Ray Developer Guide*. https://docs.aws.amazon.com/xray/latest/devguide/aws-xray.html Accessed: 2025-01-16.

Anderson. 2023. Amazon internal case study raises eyebrows. https://devclass.com/2023/05/05/reduce-costs-by-90-by-moving-from-microservices-to-monolith-amazon-internal-case-study-raises-eyebrows

Bassil and Keller. 2001. Software visualization tools: survey and analysis. In *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*. 7–17. https://doi.org/10.1109/WPC.2001.921708

Biswal. 2023. Visualization Tools: A Summary. https://www.simplilearn.com.cach3.com/tutorials/tableau-tutorial/what-is-tableau.html Last updated on January 13, 2023.

Canfora and Cimitile. 2001. Software Maintenance. *Handbook of Software Engineering*

and Knowledge Engineering 1 (01 2001). https://doi.org/10.1142/9789812389718_0005

Cosma. 2010. Reverse engineering object-oriented distributed systems. 1 – 6. https://doi.org/10.1109/ICSM.2010.5609716

Digital.ai. 2023. Exploring Reverse Engineering: Benefits, Misuse, and the Role of Application Hardening. https://digital.ai/catalyst-blog/exploring-reverse-engineering-benefits-misuse-and-the-role-of-application-hardening/ Originally Published: June 16, 2023 — Updated on March 20, 2024. Accessed: 2025-01-24.

Folmer, van Gurp, and Bosch. 2005. Engineering Human Computer Interaction and Interactive Systems. In *Engineering Human Computer Interaction and Interactive Systems*, Rémi Bastide, Philippe Palanque, and Jörg Roth (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 38–58.

Gluck. 2020. Introducing Domain-Oriented Microservice Architecture. https://www.uber.com/en-CA/blog/microservice-architecture/ Accessed: 2025-02-27.

Harris. 2024. Microservices vs. Monolith. https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith Accessed: 2025-01-24.

Klösch. 1996. Reverse engineering: Why and how to reverse engineer software. In *Proceedings of the California Software Symposium (CSS'96)*. Citeseer, 92–99.

Koschke. 2002. Software Visualization for Reverse Engineering. In *Proceedings of the International Workshop on Program Comprehension*. Springer-Verlag, Berlin, Heidelberg, 138–150. https://doi.org/10.1007/3-540-45875-1_11

Koschke. 2003. Software Visualization in Software Maintenance, Reverse Engineering, and Reengineering: A Research Survey. *Journal on Software Maintenance and Evolution* 15 (03 2003), 87–109. https://doi.org/10.1002/smr.270

Microsoft. 2024. *Azure Monitor Documentation*. https://learn.microsoft.com/en-us/azure/azure-monitor/overview Accessed: 2025-01-16.

Microsoft - Azure Monitor Best Practices. 2024. *Azure Monitor Best Practices - Analysis and Visualization*. https://learn.microsoft.com/en-us/azure/azure-monitor/best-practices-analysis Accessed: 2025-01-16.

Monzo Engineering Team. 2024. How We Run Migrations Across 2800 Microservices. https://monzo.com/blog/how-we-run-migrations-across-2800-microservices Accessed: 2025-02-18.

Müller, Mahler, Hunger, Nerche, and Harrer. 2018. Towards an Open Source Stack to Create a Unified Data Source for Software Analysis and Visualization. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*. 107–111. https://doi.org/10.1109/VISSOFT.2018.00019

Oladipo, Francisca, Odoh, Onyemaechi, Dr. Onyesolu, and Onyesolu. 2012. Exploring the two faces of Software Reverse Engineering. *International Journal of Advanced Research in Computer Science and Software Engineering* 2 (05 2012), 367–370.

Peterlić. 2024. Spring: How to Get the Current ApplicationContext. https://www.baeldung.com/spring-get-current-applicationcontext Last updated: May 11, 2024. Reviewed by: Saajan Nagendra. Accessed: 2025-01-27.

Sire. 2024. Software Development Life Cycle (SDLC): 7 Models and 8 Phases. https://www.pulsion.co.uk/blog/software-development-life-cycle-sdlc/ Accessed: 2025-01-24.

Software Engineering Institute, Carnegie Mellon University. [n. d.]. Software Architecture. https://www.sei.cmu.edu/our-work/software-architecture/ Accessed: 2025-01-24.

Spring Framework Documentation. 2025. Introduction to the Spring IoC Container and Beans. https://docs.spring.io/spring-framework/reference/core/beans/introduction.html Accessed: 2025-03-02.

Spring Team. 2025. Spring PetClinic Microservices. https://github.com/spring-petclinic/spring-petclinic-microservices Accessed: February 14, 2025.

Statistics-Easily. 2025. What is Unified Data? Understanding its Importance. https://statisticseasily.com/glossario/what-is-unified-data-understanding-its-importance/ Accessed: 2025-01-11.

The Postman Team. 2023. What is an API Endpoint? https://blog.postman.com/what-is-an-api-endpoint/

Waseem, Liang, Shahin, Di Salle, and Márquez. 2021. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and*

Software 182 (Dec. 2021), 111061. https://doi.org/10.1016/j.jss.2021.111061