THE SINGLE SOURCE OF TRUTH SYSTEM

THE SINGLE SOURCE OF TRUTH PARADIGM AS A TOOL FOR SUPPORTING SOFTWARE MAINTENANCE

By STEPAN BRYANTSEV, BS

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree Master of Applied Science

McMaster University
MASTER OF APPLIED SCIENCE (2025)
Hamilton, Ontario, Canada (Computing and Software)

TITLE:	The Single Source of Truth Paradigm as a Tool for Supporting Software Maintenance
AUTHOR:	Stepan Bryantsev BS (Software Engineering), Higher School of Economics, Moscow, Russia
SUPERVISOR:	Sebastien Mosser
NUMBER OF PAGES:	xiii, 64

Lay Abstract

As software systems grow, they often become harder to manage, with problems like slow performance, bugs, security issues, and outdated parts. Developers use different tools to find and fix these issues, but each tool gives information in its own way, making it hard to see the full picture. This project introduces a system called the Single Source of Truth (SST) that brings all this information together in one place. It organizes the data as a unified graph representation, ensuring data validations and consistency.

Abstract

Many software systems become complex over time and eventually become harder to maintain. They often face performance problems, security risks, outdated dependencies, bugs, and other issues. To address these challenges, practitioners use various maintenance tools like performance profilers, static analyzers, security scanners, and more. However, the data from these tools is often scattered and difficult to combine, making it hard to get a complete picture, perform analysis, and make informed decisions.

We introduce the implementation of the Single Source of Truth (SST) paradigm, which allows us to bring all software maintenance data together in one place. The SST aggregates information from different tools, structures it, and stores it in a consistent and reliable way. It uses a graph-based approach to organize and unify the data, making it easier to explore and analyze. The system was tested on several software projects and showed that it can help better understand the software systems and support smarter maintenance decisions.

To my family.

Acknowledgements

I would like to sincerely thank my supervisor, Dr. Sébastien Mosser, for his limitless help, guidance, and encouragement throughout this project. Without him, I would not have been able to make this journey. His advice, feedback, and support were valuable at every stage of this work.

I also want to thank my colleagues and fellow researchers at McMaster University for their collaboration and helpful discussions during the process.

Finally, I am very grateful to my family and friends for their constant support, patience, and understanding throughout this journey.

Table of Contents

Lay Abstract **iii**

Abstract **iv**

Acknowledgements **vi**

Notation, Definitions, and Abbreviations **xii**

Declaration of Academic Achievement **xiv**

1 Introduction **1**

1.1 Motivation 1

1.2 Research Goal and Questions 2

1.3 Proposed Solution 3

1.4 Thesis Structure 3

2 State of the art **5**

2.1 Software maintenance 5

2.2 Maintenance tools 7

2.3 Heterogeneous data 8

2.4 Problem definition 11
2.5 Conclusion 12

3 Solution **13**

3.1 Solution Requirements 13
3.2 Single Source of Truth Approach 14
3.3 Design of the Single Source of Truth system 16
3.4 Conclusion 17

4 Design & Implementation **19**

4.1 SST System Overview 19
4.2 Design Decisions 21
4.3 Conclusion 36

5 Evaluation **37**

5.1 Use case 37
5.2 Microservices Use Case 51
5.3 Large System Use Case 54
5.4 Limitations 56

6 Conclusion **57**

6.1 Results 57
6.2 Discussion 58
6.3 Future Work 59

List of Figures

3.1 General architecture of the proposed solution 18

4.1 System architecture. 20

4.4 Graph merging 31

5.1 Types used in the performance probe. 39

5.2 Types used in the performance probe. 40

5.3 Example of performance data in the system. 41

5.4 Types used in the code structure probe. 42

5.5 Example of code structure data. 43

5.6 Types used in the author contribution probe. 43

5.7 Example of author contribution data. 44

5.8 Global type schema after probe registration. 45

5.9 Merged global graph representing the integrated system data. 46

5.10 Slowest methods and their locations. 47

5.11 Methods invoking slow methods. 48

5.12 Contributors of slow methods, responsible for potential optimizations. 49

5.13 Visualization of author contributions based on lines of code edited. . 50

5.14 Execution time of methods categorized by self-time, I/O operations, and outgoing calls. 51

5.15 Contributions of authors to slow methods, helping identify the best candidates for optimization tasks. 52
5.16 Example of a merged graph 53

List of Tables

3.1 Comparison of data integration approaches. 15

5.1 Node and relationship counts in the Petclinic project 44

5.2 Node and relationship counts in the NetBeans dataset 55

Notation, Definitions, and Abbreviations

Notation

$G = (V, E)$	A graph G consists of a set of nodes V and a set of edges E
$v(p)$	The value of property p in node v
$v \equiv v'$	Nodes v and v' are considered equivalent and can be merged
$G_1 \otimes G_2$	Merge operation of two graphs G_1 and G_2
$\cup$	Set union operator

Definitions

Probe	A data provider that sends maintenance data to the SST system
Graph Merging	The process of combining graphs from multiple probes by merging equivalent nodes and edges

Type Equivalence	A rule that defines when two nodes of the same type are considered identical for merging
Global Schema	The combined type schema used by the SST system to validate and integrate all incoming data
Data Abstraction	A method of simplifying complex data by showing only the relevant information while hiding low-level details, making it easier to understand and analyze

Abbreviations

SST	Single Source of Truth
ETL	Extract, Transform, Load
AI	Artificial Intelligence
ML	Machine Learning
API	Application Programming Interface
BI	Business Intelligence
DSL	Domain-Specific Language

Declaration of Academic Achievement

This thesis is my original work and reflects my independent effort and research. I, Stepan Bryantsev, contributed to the study, design, and implementation of the proposed system. The work was conducted under the supervision of Dr. Sébastien Mosser, who provided guidance, feedback, brainstorming support, and overall direction throughout the process.

Mina Mahdipour, Waqar Awan, and Hassan Zaker contributed to the development of certain components used during the implementation phase. All contributions have been clearly acknowledged, and all external sources of information have been properly cited.

Chapter 1

Introduction

1.1 Motivation

Software maintenance is a critical part of the software development process. After a system is delivered, it needs to be updated, improved, and adapted over time. In many cases, maintenance takes more time and resources than the initial development. But it is not only about cost—software systems are used in areas like healthcare, transportation, finance, and infrastructure, where failures can impact people's lives and safety [22]. In such cases, proper maintenance becomes essential to ensure that systems stay reliable and trustworthy over time.

Now imagine working with a large software project that has been developed and maintained for several years. Over time, the project has become complicated: there might be code quality issues, performance slowdowns, unpatched security vulnerabilities, tangled dependencies, and other problems. To deal with this, you need to use multiple tools—such as performance profilers, code analyzers, security scanners, and dependency checkers. Each tool produces different types of data in different formats,

which makes it hard to store, integrate, and analyze all this information together.

This situation creates a clear need for a system that can bring all the maintenance data together. The solution must be able to collect data from many different tools, organize it in a consistent way, and make it easy to use for decision-making. While there are other possible approaches—like data warehouses and lakes, ML and AI-based integrations, knowledge driven integrations, manual spreadsheets, or custom tool scripts—these often focus on specific problems, lack transparency, are hard to scale, or require too much manual effort. What is missing is a general, trustworthy system that helps integrate heterogeneous data in a structured and reliable way. This work explores such a system and proposes a solution based on the Single Source of Truth (SST) approach.

1.2 Research Goal and Questions

This work is based on the idea of using the Single Source of Truth paradigm to improve the way we manage software maintenance data. SST is a common approach in data-driven systems that helps ensure consistency by storing all relevant information in one trusted location. It reduces duplication, avoids conflicts, and makes it easier to access and work with the data.

In this thesis, we treat SST as a high-level guideline for designing the solution. It gives a clear direction for building a consistent and reliable system, but it does not solve all the challenges by itself. To apply SST in the software maintenance domain, we need to go beyond the idea itself and make research, design, and implementation work required to build a usable solution.

In particular, we aim to answer the following research questions:

- **RQ1:** How can we organize and structure data coming from multiple tools and technologies in a single system?
- **RQ2:** How can we ensure that this data remains consistent and can be trusted for decision-making in software maintenance?

1.3 Proposed Solution

To address the challenges, we propose the implementation of the Single Source of Truth (SST) system. The main idea is to create a central, trustworthy place where all relevant maintenance data from different tools can be collected, structured, and used. The goal of the system is to make maintenance data easier to integrate, store, access, and analyze. By doing this, the system supports better decision-making and reduces the complexity of working with multiple maintenance tools.

The proposed solution provides the following key benefits:

- A way to formalize and represent diverse maintenance data from different tools and sources.
- A unified system where data is ready for querying, visualization, and analysis.
- A clear set of integration rules that ensure the consistency and transparency of the data.

1.4 Thesis Structure

This thesis is organized as follows:

Chapter 2: State of the Art Outlines the significance of software maintenance and the existing challenges in managing heterogeneous maintenance data.

Chapter 3: Solution Presents the Single Source of Truth approach and the main solution as a way to unify, integrate, and validate data from multiple sources.

Chapter 4: Design & Implementation Explains how the SST system is designed and implemented.

Chapter 5: Evaluation Demonstrates the framework in action through real-world use cases, showing how maintenance data can be combined and utilized.

Chapter 6: Conclusion Summarizes the work and suggests directions for future research and improvements.

Chapter 2

State of the art

This chapter focuses on the importance of software maintenance as a key part of the Software Development Lifecycle. It explains how maintenance tasks, like bug fixing, performance improving, refactoring, security enhancement and documentation management, are essential for keeping systems reliable, stable, and secure over time. The chapter also discusses the challenges caused by the variety of tools and techniques used during the maintenance process, particularly in managing and integrating the different data they produce. Finally, it highlights the need for a unified and standardized approach to solve these challenges and improve the way maintenance data is handled.

2.1 Software maintenance

Software maintenance is an essential process of the Software development lifecycle that ensures long-term system reliability, stability, and security [6]. IEEE standard defines software maintenance as "*Modification of a software product after delivery to*

correct faults, to improve performance or other attributes, or to adapt the product to a modified environment" [4]. It is important to understand what processes, issues, and challenges are hidden in this definition and what goals people pursue while performing software maintenance.

Software maintenance is an extremely broad field that includes such aspects as bug fixing, performance optimization, migrations, refactoring, documentation building, and many others. As a result, there are various ways to classify and formalize those tasks. One possible approach is to categorize these tasks into three groups: *adaptive, perfective, and corrective maintenance* [23]. Adaptive maintenance focuses on adapting software to changes in its environment, perfective maintenance involves enhancing functionality or performance, and corrective maintenance addresses bug fixes and fault corrections. This approach helps to understand and classify the types of changes and their impact on the target project.

Speaking about the software maintenance process, it contains multiple phases, including change management, analysis, design, implementation, testing, and delivery. All these phases require comprehensive knowledge and understanding of the system to ensure successful analysis of the project and reduce maintenance costs [11]. According to Gupta et al. [8], key challenges in software maintenance include high costs, difficulties in tracking the impact of changes, and the time-intensive nature of understanding code.

Overall, software maintenance is a very large and important phase of the Software Development Lifecycle that ensures a project's life after delivery. This process contains wide range of challenges and requires comprehensive knowledge and analysis from different perspectives and different points of view.

2.2 Maintenance tools

Software maintenance involves a wide range of tasks, which has led to the development of various tools to support the process. A key focus in this domain is extracting knowledge from the project. Such tasks are often achieved using various software analysis or reverse engineering tools. Valentina Lenarduzzi et al. [12] created a detailed survey analyzing 25 different software analysis tools, providing insights into their goals, supported technologies, and trends. The classification covered objectives such as code review, bug detection, testing, identifying security flaws, and others. Furthermore, the study highlighted that researchers and practitioners often have different needs, which further broadens the variety of tools available.

In general, maintenance tools offer a wide range of solutions designed to support various aspects of software and project management. These include reverse engineering, software analysis, and many other tools that support various aspects of software maintenance. *Reverse engineering tools* aim to extract high-level abstractions, such as system architecture diagrams, dependency graphs, or functional overviews, from software, particularly when documentation is incomplete or unavailable. While *software analysis tools* focus more on evaluating code quality, identifying issues such as bugs, security vulnerabilities, or performance inefficiencies.

Further, these tools can be divided into *static* and *dynamic* analysis tools. Static analysis tools operate without executing the code, providing insights into syntax, structure, and potential compile-time issues. On the other hand, dynamic analysis tools analyze a program's behavior during execution, offering various runtime insights [21, 3].

In practice, sometimes, even selecting appropriate tools for a specific project can

be a challenging task due to the wide variety of available solutions. To address this challenge, Gerald C. Gannod and colleagues [7] developed a framework to analyze and structure reverse engineering tools based on their semantic quality. This framework helps in comparing different approaches by measuring and formalizing the characteristics of existing tools. Such efforts provide valuable guidance in identifying tools that suit the best for particular project goals.

2.3 Heterogeneous data

As projects grow in complexity, relying on a single tool is often not enough. This raises an important question: *what if more than one tool is needed for project maintenance?* For example, what if the process has to address both performance optimization and bug fixing simultaneously. Each task may require specialized tools, resulting in separate sets of data and insights. How can all this information, coming from different sources, be effectively managed, and utilized to support maintenance activities? Martínez-Fernández et al. [13] present an ontology-based approach to integrate data from different software analytics tools, such as static code analysis, testing tools, and issue trackers. This work demonstrates how automation and data comprehension can help reduce the effort required to manage and integrate data, making it more useful for software maintenance decisions.

Many researchers conclude that the usage of maintenance tools in software projects is a challenging problem that needs a systematic approach [2]. Pfeiffer and Aaen identify the difficulties of using multiple tools for software quality monitoring, emphasizing issues such as inconsistent data formats, lack of synchronization, and the additional burden on practitioners to align tool outputs for effective decision-making [17]. These

challenges highlight the need for efficient strategies to aggregate and integrate the outputs of multiple maintenance tools and data sources, ensuring that the maintenance process remains effective and manageable.

The next question is: "*what are the actual technical and practical solutions that address the problem of heterogenous data integration*". Exist a significant number of solutions and approaches that helps to integrate data during the software maintenance process, however we can highlight 3 the most popular and large groups.

Data driven integration This is a very large group of solutions that mostly focus on collecting, transforming and storing data in a structured way. Usually, such approaches rely on the data properties and require significant understanding of the domain, data formats and shapes during the implementation. The group includes such solutions as ETL (Extract, Transform, Load) and Data Warehousing [10]; Data Lakes [9]; various relational databases techniques; graph solutions and many other. For example, some approaches use graph-based models to combine syntactic and semantic information, allowing efficient querying and analysis to support tasks like debugging and refactoring [19]. Other approaches focus on automation in data aggregation process. They reduce the need for manual documentation and ensure stakeholders have access to accurate and up-to-date information. Buchgeher et al. introduce a platform for automatically extracting and maintaining architecture information for large-scale service-oriented software systems [1]. Generally speaking, such data-driven solutions are great in terms of data management and consistency, however, could be too tailored to specific problem and experience lack of flexibility and scalability.

Machine Learning & AI approaches Nowadays, ML and AI solutions become

more and more popular for all spheres of Software development, and maintenance process is not an exception. These solutions include automated AI-based data integrations, ML data cleaning, anomaly detections, AI-Driven schema mapping and many other. The solutions tackle the problem by identifying patterns and issues using predictive models and evolutionary approaches [5]. Overall, ML and AI contains very powerful solutions that help automate, optimize, and simplify the handling of heterogeneous data in software maintenance. Such solutions are great in terms of automation, efficiency personalisation, however, they may lack control, transparency, and high-level abstraction. In addition, many AI solutions work as a black box so its hard to understand the processes and hence such solutions could be less trustworthy. Despite their strengths, AI-based solutions often need to be complemented by more transparent and controllable approaches.

Knowledge driven integration Knowledge-driven integration uses formal representations of domain knowledge—such as ontologies, rules, and semantic models—to support and improve the integration of maintenance data. Unlike data-driven or AI-based methods, this approach incorporates expert knowledge and software concepts, allowing the system to reason about relationships and provide greater transparency [24]. Rule-based reasoning can help detect inconsistencies, validate tool outputs, and guide maintenance decisions. A great example is the DEFII framework [20], which introduces an ontology-based tool for integrating data from engineering design and analysis models. Overall, knowledge-driven integration offers strong benefits in terms of structure, clarity, and explainability, However, building and maintaining ontologies and formal rules can be

time-consuming and often requires specialized domain expertise.

2.4 Problem definition

The use of multiple maintenance tools often creates significant challenges in the areas of data integration, aggregation, and consistency. This is due to the wide variety of solutions available and the heterogeneous nature of the software maintenance domain [2]. Pfeiffer and Aaen emphasize that combining outputs from multiple tools is a complex task, as it involves inconsistent data formats, lack of synchronization, and additional work for practitioners to align outputs for decision-making [17]. These difficulties underline the necessity of developing efficient strategies to aggregate, integrate, and analyze outputs from diverse maintenance tools, ensuring the process remains manageable and effective.

After examining existing challenges and solutions in the field, we can identify three global weaknesses and areas of improvement in current approaches:

Focus on Specific Tools and Technologies [**C1**] A significant limitation of existing software maintenance solutions is their narrow scope. Many approaches are designed to work only with specific programming languages, tools, or frameworks. For example, static analysis solutions might support only a few programming languages and integrate with limited number of technologies. As a result, there is a clear need for a universal approach that can manage a broader range of technologies and tools.

Lack of Formalization and Abstraction [**C2**] Many existing solutions focus on data features and rely on integration at the data level. While these approaches

can achieve some success, they often lack consistency and fail to provide meaningful high-level abstractions. This makes it difficult to understand and analyze the nature of the data or the processes involved. A formalized, abstraction-driven approach is necessary to enhance the traceability, adaptability, and comprehension of the data flow in maintenance activities.

Trust and Transparency in Data Processes [**C3**] The final global challenge is ensuring that the aggregated data is trustworthy. It is essential to comprehend and control the data sources, the data structure, the integration and storage processes. Such transparency helps to ensure that all processes are under control and that the data is consistent, accurate, and reliable. Trust in the data is a foundation for effective decision-making and must be addressed in a comprehensive solution.

2.5 Conclusion

The described challenges highlight the need for a universal, formalized, and trustable approach to addressing heterogeneous data in software maintenance. Current solutions fail to meet the requirements for handling diverse data across multiple tools and technologies, integrating and analyzing that data at a high level, and ensuring the reliability and traceability of the information. *There is a need to study an approach that supports data aggregation, integration, analysis, and management while maintaining consistency and trust.*

Chapter 3

Solution

This chapter introduces the SST system. It is designed to solve the challenges described in the previous chapter by helping practitioners collect, organize, and use maintenance data from different tools in one place. SST focuses on making the data integration consistent, clear, and easy to work with.

3.1 Solution Requirements

The main goal of the solution is to support practitioners in software maintenance tasks by providing a trustworthy system that helps ensure data consistency and reliability for a wide range of tasks and tools. Such an approach can benefit software project analysis, decision-making, and maintenance across diverse environments and over time. To meet these goals and address the challenges outlined in Section 2.4, the system must satisfy the following requirements:

- **Multiple maintenance solutions:** The solution must handle data provided by multiple maintenance reverse engineering and software analysis tools without

being tied to specific tools or technologies (Challenge C1).

- **High level abstractions:** The solution must rely on high-level data abstractions and allow users to specify the data formats and schema (Challenge C2).
- **Data aggregation and integration:** The solution must aggregate data from various sources and integrate it into a unified representation for storage and analysis (Challenge C3).
- **Data validation and consistency:** The solution must validate all input data to ensure consistency and correctness (Challenge C3).
- **Accessibility and availability:** The solution must allow multiple practitioners to work concurrently, providing access to data for updates, analysis, and decision-making (Challenge C1).
- **Time evolution and environments:** The solution must support data evolution over time and support multiple working environments (Challenge C3).

3.2 Single Source of Truth Approach

The initial idea and implementation of the system were inspired by the Single Source of Truth approach, which is widely used in various organizations and solutions. The Single Source of Truth approach focuses on improving data-driven decision-making by ensuring consistency, accuracy, and reliability of data. As defined by Magno Queiroz et al., *"The goal is to have a single definitive source of data, accessible, trusted, credible, and reliable"* [18].

The Single Source of Truth approach addresses common issues such as data silos, where information is isolated in separate systems, and inconsistencies caused by duplicate or outdated data. By consolidating data into one trusted source, Single Source of Truth helps reduce errors, simplify operations, and make data easier to use. It also supports scalability and adaptability, enabling organizations to manage growing data needs and respond to changes effectively.

Another key benefit of the Single Source of Truth approach is its ability to promote collaboration and data sharing across users and teams, ensuring everyone works with the same accurate information. According to the paper [18], organizations that implement Single Source of Truth report higher efficiency, improved trust in their data, and better organizational flexibility. These strengths make the approach an ideal foundation for solutions that depend on reliable maintenance data.

Table 3.1 summarizes how well different integration approaches address the described challenges.

Problem	**Data-driven integration**	**AI & ML solutions**	**Single Source of Truth**
Variety of supported tools and technologies	X	✓	✓
Data formalisation and abstraction	✓	X	✓
Trust and Transparency	X	X	✓

Table 3.1: Comparison of data integration approaches.

As a result of the analysis, we decided to use the Single Source of Truth as the foundation for the solution's implementation. However, it does not mean that other data integration, AI, ML, and other techniques could not be part of the system or are

not suitable for the problem. In the future, various solutions could be incorporated to improve the system's usage, performance, flexibility, analysis, and other processes, but for the initial and fundamental implementation, we mostly rely on the Single Source of Truth approach.

3.3 Design of the Single Source of Truth system

Building on the concept of the Single Source of Truth approach, we designed our SST system to provide two main benefits for software maintenance:

- **Single Source:** The system acts as a centralized storage for all maintenance data, ensuring that all data is gathered in one place. This is achieved through a formalized integration method that combines diverse sources into a single accessible system.
- **Truth:** The system is responsible for aggregating and integrating data in a transparent and reliable way, ensuring the accuracy and consistency of the information. This is ensured by defining the data formats and schema and using a formal and strict way of data integration.

To implement these principles, the SST architecture is organized into three conceptual layers (Figure 3.1):

- **Maintenance Tools (Probes) Layer:** This layer represents the set of tools used for software maintenance processes. These tools act as the primary sources of data, which they deliver to the framework in a standardized graph-based format.

- **SST service Layer:** The core of the solution, this layer is responsible for integrating the data received from the maintenance tools. By organizing information into a unified graph representation, the SST service ensures that the data is stored in a consistent way and is ready for further analysis.

- **Data Access Layer:** This layer provides practitioners with the capability to plug in different data visualization or analysis tools. The architecture does not rely on a single representation and analysis technique, but provides flexibility.

The primary goal of the SST design is to benefit software maintenance tasks by providing a central system for managing and analyzing data from heterogeneous tools. In simple terms, SST can be imagined as a powerful and universal adapter, hub, and data storage that, on one side, allows the integration of diverse data sources into a single, reliable representation, and, on the other side, allows practitioners to perform data warehousing and analysis.

3.4 Conclusion

This chapter introduced the Single Source of Truth approach as the foundation for addressing the key challenges of software maintenance data integration. The approach fulfills solution requirements by consolidating diverse maintenance data into a centralized, consistent, and transparent system. By implementing this approach, we can build a flexible and reliable system that enables better understanding, analysis, and decision-making in software maintenance tasks across different tools.

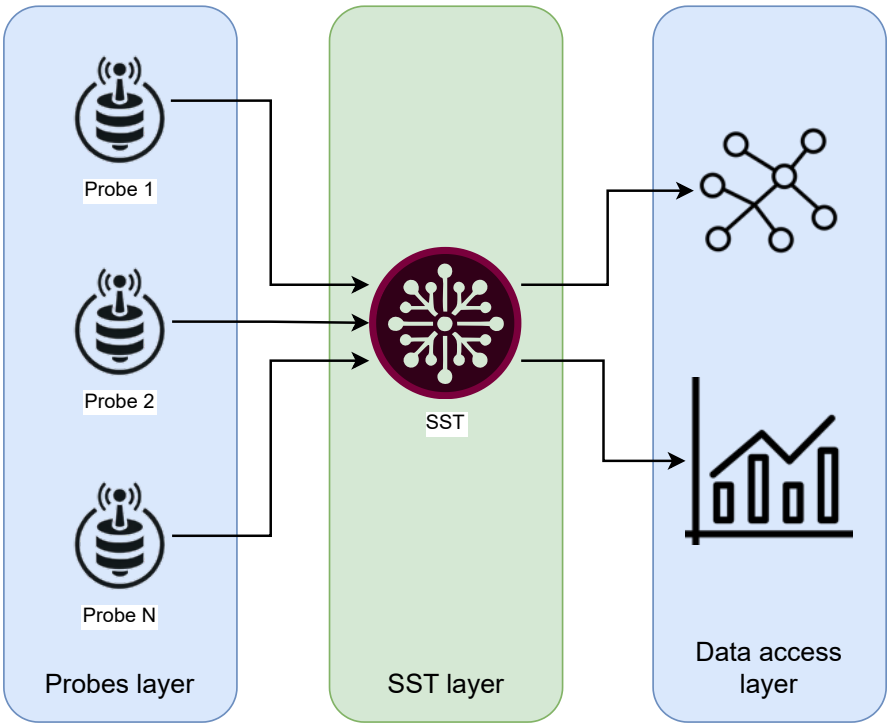

Figure 3.1: Architecture of the solution

Chapter 4

Design & Implementation

This chapter presents the core design and implementation of the SST system. The goal is to create a reliable and consistent framework for integrating heterogeneous data from multiple maintenance tools into a unified representation that supports analysis and decision-making.

4.1 SST System Overview

The data flow starts from the software maintenance tools, that could include diverse data structures and formats. To be compatible with the SST each data source must be represented as *probe* and follow strict format and schema rules. First each probe is required to provide data schema definition to ensure compatibility with other probes. After that it is allowed to push the data to the remote SST storage.

The SST service is implemented as a remote web server. Such centralized design ensures that users are not constrained by a single machine or setup. This flexibility is particularly valuable in maintenance scenarios that often require diverse and

resource-intensive environments. As the result, all probes can push their data to the centralized SST server, ensuring that the data is aggregated and accessible in a single, consistent location. In addition, the SST provides powerful integration mechanism that prevents data duplication and allows to independent data upload. As the result of integration, the SST system operates with consistent and unified graph representation of all incoming data. Finally, once the data is stored the SST system provides a layer for data access that could include data visualisation, representation, and analysis techniques.

Figure 4.1 provides a high-level overview of the system architecture. The SST system follows a layered architecture. The REST API controller receives and validates all incoming HTTP requests. The services layer applies the main logic for handling and organizing Probes, and data. Finally, the infrastructure layer provides commands and queries to access the Neo4j graph database, ensuring efficient storage and retrieval of information.

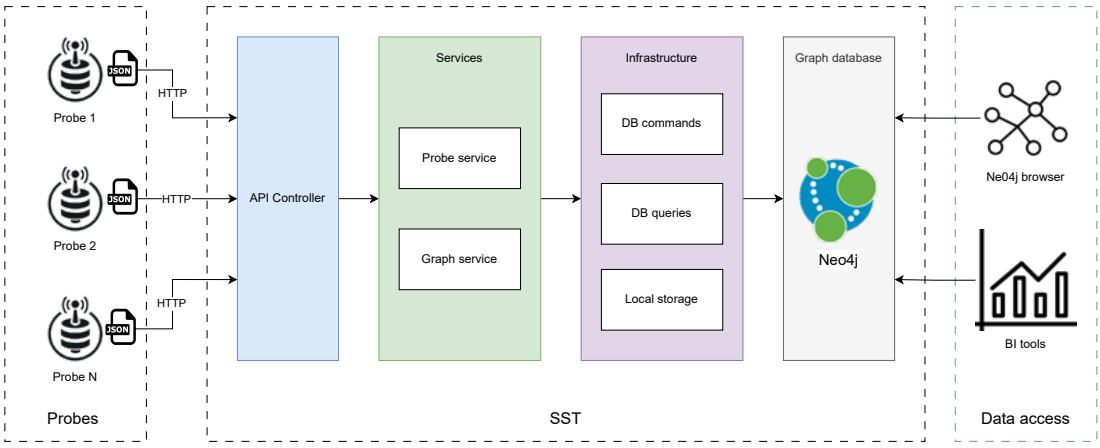

Figure 4.1: System architecture.

4.2 Design Decisions

Several key design decisions were made during the development of the SST system. These form the core of the project and ensure it meets the requirements outlined in previous chapters. This section describes each design solution and the addressed challenges to better understand the main contribution of the work.

1. **Probes** How external data sources integrate with the SST.
2. **Type System:** How data is structured and formalized.
3. **Graph Storage:** How the system represents data
4. **Graph Merging:** How data is combined and aggregated consistently.
5. **Data Access:** How practitioners interact with and analyze the data.

4.2.1 Probes

Challenge: How to support heterogeneous data sources?

Design

Imagine having multiple maintenance tools providing different data. It could be performance, code structure, runtime, security, developer contribution, and other data of different shapes and formats. It is impossible to tailor the solution for each possible case, so the challenge is to create the appropriate generalization that will allow support for a wide range of data sources.

To address this challenge, we introduce the notion of Probes. A Probe represents a maintenance tool that provides data to the SST framework. By defining this abstraction, we establish a separation between data providers and the framework itself and enable the ability to develop and integrate new data providers.

This approach supports the solution's flexibility and scalability by allowing users to plug in and unplug different tools, reducing the integration implementation effort. Such a design ensures that a wide range of tools can be supported, enabling the system to aggregate data from diverse sources.

Each Probe has only two requirements:

- **Provide types schema** (4.2.2).
- **Provide data in a graph format** (4.2.3).

As a result, we have a formal approach that allows us to integrate any maintenance data provider into the SST in the proper format. A long-term ambitious goal of this design is to create a marketplace of Probes, where all practitioners can contribute and create a pool of tools compatible with the SST. Such an ecosystem would function similarly to a package manager, enabling seamless integration of new tools while managing their compatibility.

Implementation

To support this abstraction, the SST system implements a structured process for registering and validating probes. The following steps outline how probes are onboarded and made compatible with the global schema.

Each probe must be registered in the system and define its own data schema. When a new probe sends a registration request, the SST follows a structured validation and integration process to ensure consistency and compatibility of the provided types.

The probe registration process consists of the following steps:

1. **Probe validation.** The SST system validates the probe definition. The system checks whether a probe with the same name already exists and verifies that all data types within the probe are unique and defined correctly. Additionally, it ensures that each type has a valid structure, including a mandatory definition of the merge rule (type equivalence).

2. **Probe compatibility.** If the probe defines a data structure that is already present in the system, the algorithm compares the type structures. It verifies that the fields and merge rules of the new type are compatible with the existing definition. If the types are not compatible, the probe registration is rejected to maintain consistency in the system.

3. **Updating the global schema.** After processing the local data schema of a probe, the SST updates the global schema. This schema serves as the global data structure for validating future data submissions from probes.

This process ensures that all probes conform to a consistent structure, preventing conflicts and inconsistencies in the stored data. By enforcing type compatibility and merging definitions when possible, the SST system creates a *unified schema*. This schema plays a key role in validating the data received from probes and maintaining data consistency across the SST.

4.2.2 Type System

Challenge: How to define data abstractions and schemas?

Design

To successfully integrate and manage input data from multiple sources, we need to understand its formats and shapes. Instead of working directly with the data, the SST requires providing the Type schema. Such an approach ensures better data abstraction and generalization, allowing improved data integration, validation, and consistency. The Type System is a critical component between the Probes and the SST. Each Probe is required to define the types of data it provides, ensuring that the framework can understand, process, and integrate the data correctly.

Imagine having a type that represents a data schema. Like a class in OOP, the type defines the fields that the future data component (object) will have. In addition, the Type System allows defining the relations between different types inside one probe and also defines type equivalence globally. For example, there is Probe 1 that contains Type A and Type B, and another Probe 2 that contains Type A' and Type C (Figure 4.2).

To successfully implement data integration and compatibility, the SST requires each Probe to define Type equivalence for each type. So, when the two probes that share some data types are integrated in the SST, the shared data Types are also integrated. As a result, the global Types will contain Type B, Type C, and the integrated Type A, which will contain the properties from both Probes as long as there are no conflicts (Figure 4.3). By establishing this approach, the SST can ensure that the data at the schema level is compatible and will be integrated correctly.

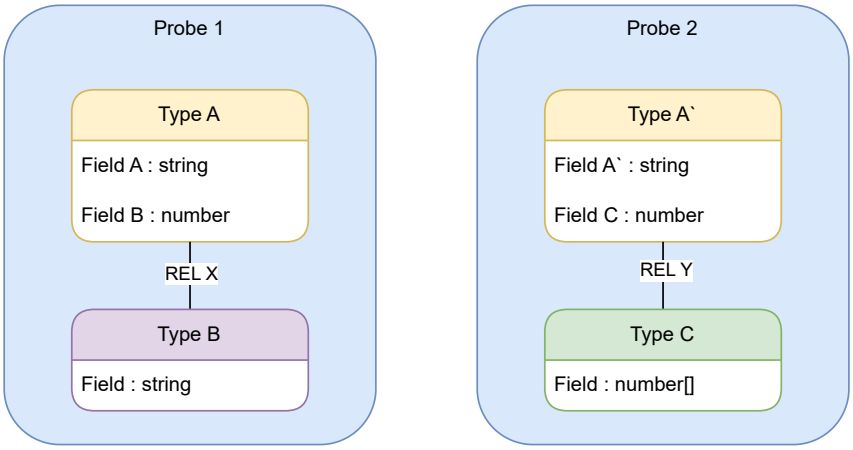

Figure 4.2: Type schemas of two probes

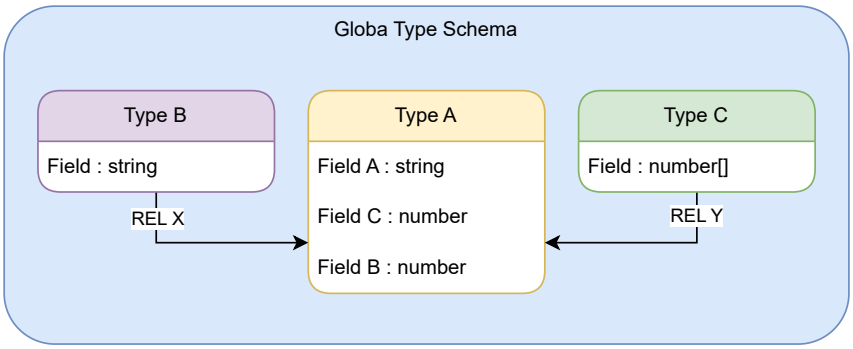

Figure 4.3: Global Type System

One significant advantage of the Type System is the creation of a global type system (schema), which provides a high-level abstraction of the data structures and relations. This abstraction enhances understanding of the aggregated data and improves usability, especially if the number of Probes and types grows and the relations become more complex.

Implementation

Based on the design goals, the SST system requires each probe to declare its data schema using a structured JSON format. This ensures type definitions are explicit, consistent, and mergeable across different probes. The expected JSON structure consists of the following elements:

- **types**: A list of types used in the Probe, where each type has:
  - **name**: The unique name of the type.
  - **fields**: A list of attributes describing properties of the node, including:
    - **name**: The field's name.
    - **type**: Expected data format.
    - **unique**: A boolean flag indicating whether the field is unique.
  - **mergeRules**: Specifies the type equivalence (how nodes of the same type should be merged).
- **relations**: A list of relationships between types:
  - **name**: The name of the relationship.
  - **from**: The source type of the relation.
  - **to**: The target type of the relation.

An example JSON structure for defining types is provided below:

```
{
  "typeExample": {
      "name": "fullName",
      "type": "string",
      "unique": true
```

```
6     },
7     "relationExample": {
8         "name": "RELATION",
9         "from": {
10            "typeName": "Author"
11        },
12        "to": {
13            "typeName": "Method"
14        }
15    }
16 }
```

This schema ensures that different Probes follow a standardized format while allowing flexibility for diverse data sources. Basically each type schema contains a list of data structures and a list of possible relations between those structures. Such approach is helpful when trying to formalise and structure heterogeneous and unstructured data.

4.2.3 Graph Data Format

Challenge: What data structure to choose to store SST data?

Design

There are several options for how the SST could store and manage the data. One option is to use a relational database. Such an approach is great for structured data but could be ineffective when dealing with heterogeneous and evolving information. Another option is to use a NoSQL solution that would offer much more flexibility and can support unstructured and evolving data. However, such an approach could be less convenient when working with interconnected data with multiple relations or dealing with data consistency. Given the heterogeneous nature of the maintenance

data and the need to support various data relations, we naturally end up using a graph data storage. Graph databases provide a more natural and efficient way to manage such data by focusing on relationships, making them the best choice for this problem.

Graph storage has several key advantages, making it well-suited for the SST's goals. First, it natively supports relationships, eliminating the need for complex integration operations. Second, it is flexible, allowing the SST to evolve as new data sources and tools are added. Third, querying interconnected data is efficient and intuitive, enabling complex queries that provide valuable insights during the analysis process. Lastly, graph storage aligns perfectly with the SST approach, ensuring that maintenance data is aggregated, consistent, and easy to explore, making it the most natural and effective data structure for the solution.

Implementation

After defining the types, probes can submit their actual data. To align with the graph-based model, this data must follow a specific JSON format representing two lists of nodes and relationships:

- **nodes**: A list of nodes where each:
  - **type**: Matches a previously defined type.
  - Contains additional fields based on the type definition.
- **edges**: A list of relationships between entities:
  - **relationName**: Matches a previously defined relation type.
  - **from**: The source entity (defined by type and unique identifier).

- **to**: The target entity (defined by type and unique identifier).
- **properties**: Additional attributes related to the relationship.

The following JSON example illustrates the structure of actual graph data:

```
{
    "probeName": "AuthorContribution",

    "nodes": [
      {
        "type": "Author",
        "email": "john.doe@example.com",
        "fullName": "John Doe"
      },
      {
        "type": "Method",
        "fullName": "com.example.MyClass.myMethod()"
      }
    ],

    "edges": [
      {
        "relationName": "CONTRIBUTES",
        "from": {
          "nodeType": "Author",
          "propertyName": "email",
          "propertyValue": "john.doe@example.com"
        },
        "to": {
          "nodeType": "Method",
          "propertyName": "fullName",
          "propertyValue": "com.example.MyClass.myMethod()"
        },
        "properties": {
          "linesEdited": [10, 15, 20],
        }
      }
    ]
}
```

4.2.4 Graph Merging

Challenge: How to integrate multiple graphs while ensuring consistency?

Design

A key feature of the SST graph storage is the merging principle. Using type-level equivalence defined by the Probes, the framework identifies and merges equivalent nodes. If two graphs provide equivalent nodes, in the resulting graph the equivalent nodes will be merged (Figure 4.4).

Such a process ensures two critical properties:

1. **Data Integration:** When two Probes provide graphs with overlapping data, the merging operation links their graphs by combining equivalent nodes.

2. **Data Consistency:** The merging operation eliminates duplicate nodes, ensuring that the data remains unified and coherent.

The merging principle also introduces useful *commutativity* and *associativity* properties. Regardless of the order in which data is pushed to the system, the resulting graphs will be identical. This property significantly enhances the system's usability, especially in incremental data integration scenarios where information is extended over time.

To prove Associativity and Commutativity, we can define each graph as a set of nodes and edges:

$$G = (V, E),$$

where V is the set of nodes and E is the set of edges. The merge operation for graphs $\otimes$ is defined as the union of the node sets and edge sets, with the special rule that

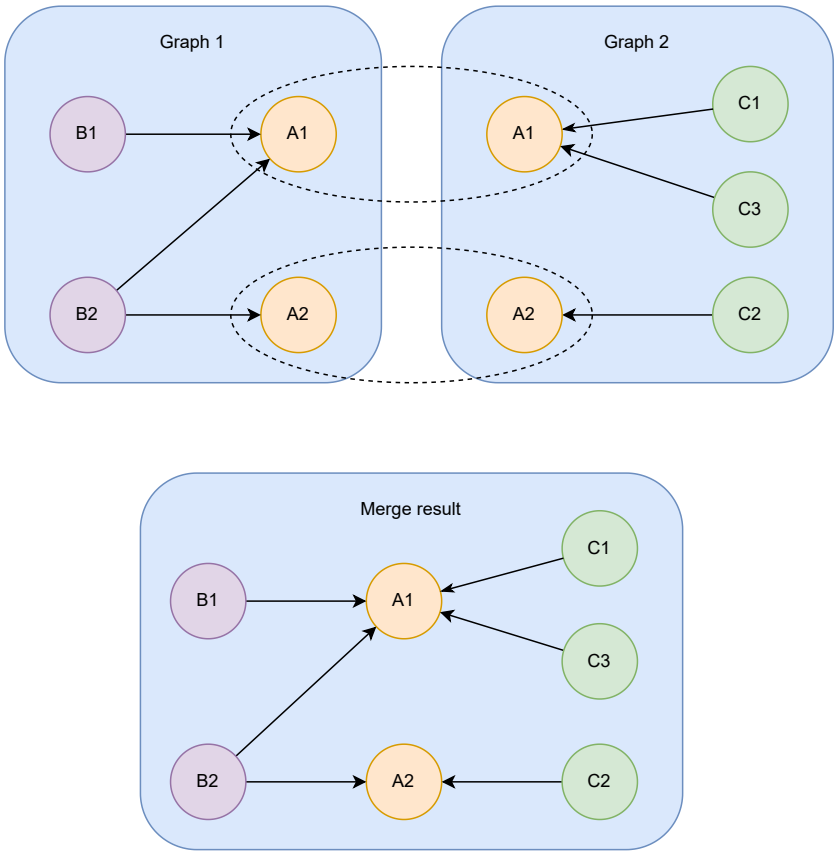

Figure 4.4: Graph merging

merges equivalent nodes. That is, for two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, we have:

$$G_1 \otimes G_2 = \Big(\big(V_1 \cup V_2\big),\ E_1 \cup E_2 \Big),$$

where $\big(V_1 \cup V_2\big)$ is the union of nodes in which any two nodes $v \in V_1$ and $v' \in V_2$ with $v \equiv v'$ are merged into a single node.

When merging two equivalent nodes, we can represent each node as a set of

property–value pairs. For example, a node v can be represented as:

$$v = \{(p_1, a_1), (p_2, a_2), \ldots, (p_n, a_n)\}.$$

The merge operation of equivalent nodes is defined as:

$$v \otimes v' = \begin{cases} v \cup v', & \text{if } \forall p \in \text{prop}(v) \cap \text{prop}(v'),\ v(p) = v'(p), \\ \text{reject}, & \text{otherwise}, \end{cases}$$

where $\text{prop}(v)$ is the set of properties present in v and $v(p)$ is the value of property p in v. Since the union of sets is both commutative and associative, it follows that:

$$v \otimes v' = v' \otimes v,$$

$$v \otimes (v' \otimes v'') = (v \otimes v') \otimes v'',$$

ensuring that the merging of nodes is a commutative and associative operation.

Graph merge commutativity

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. Then the merged graph is given by:

$$G_1 \otimes G_2 = \Big(\big(V_1 \cup V_2\big),\ E_1 \cup E_2\Big).$$

Since set union is commutative ($V_1 \cup V_2 = V_2 \cup V_1$ and $E_1 \cup E_2 = E_2 \cup E_1$) and the merge of equivalent nodes is commutative, then:

$$G_1 \otimes G_2 = G_2 \otimes G_1.$$

Graph merge associativity

Consider three graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, and $G_3 = (V_3, E_3)$. The merge operation:

$$G_1 \otimes (G_2 \otimes G_3) = \Big(V_1 \cup (V_2 \cup V_3),\ E_1 \cup (E_2 \cup E_3)\Big),$$

and by the associativity of set union,

$$V_1 \cup (V_2 \cup V_3) = (V_1 \cup V_2) \cup V_3,$$

with a similar property for the edge sets. Since the merge of equivalent nodes is also associative, it follows that:

$$G_1 \otimes (G_2 \otimes G_3) = (G_1 \otimes G_2) \otimes G_3.$$

The commutativity and associativity properties significantly enhance the usability of the SST, especially in incremental data integration scenarios where information is provided partially and over time.

Implementation

To realize the behavior defined earlier the main SST usage scenario includes two stages: *graph validation* and *graph upload*. Users are allowed and encouraged to validate the data before saving the data, to gain insights or detect data issues. The validation result includes information about created and merged nodes and edges or any errors that might cause integration or consistency issues. If the validation is successful, the SST can safely merge the graph data into the global storage. This process

guarantees consistency, prevents conflicts, and maintains a coherent representation of the information.

The algorithm consists of the following steps:

1. **Validation of schema and probe.** Before processing the graph, the system checks whether the probe that submitted the data is registered in the SST. It also verifies that the global type schema exists and is properly defined.

2. **Validation of node types.** Each node in the submitted graph is checked against the type schema. If a node references an undefined type, the submission is rejected.

3. **Grouping nodes by type.** Nodes are grouped based on their types to speed up processing. Each type is then validated against the probe definition to ensure its fields and properties match the expected structure.

4. **Merging with existing data.** Based on the type equivalence definition, the SST checks whether submitted nodes already exist in the global graph. If a node matches an existing one, the properties are compared. If the data is consistent, the node becomes a merge candidate; otherwise, an integration conflict is raised.

5. **New nodes.** If a submitted node does not match any existing entity, it is considered new and will be added to the global graph.

6. **Validation of relationships.** The system verifies that all edges (relationships) reference valid node types and properties. If an edge is not properly declared in the probe definition or links undefined nodes, an integration conflict is raised.

7. **Updating the global graph.** After all nodes and edges are validated successfully, they are integrated (merged or created) into the existing Neo4j graph database.

This algorithm ensures that data from different probes is correctly validated and integrated while maintaining consistency across the SST. If an integration conflict is raised, the data is not saved and the user receives a detailed error message with reasoning and references to the problematic data.

4.2.5 Data access layer

Challenge: How to provide flexible data access and analysis options?

Design

In software maintenance, different tasks require different ways to explore, analyze, and visualize data based on specific goals. Some may need a graph analysis approach, while others need advanced visualizations, statistics, and charts. A predefined analysis approach would limit flexibility and the number of use cases. To make the system more practical and adaptable, it is important to provide an open and customizable way for users to interact with the SST.

The Data Access design solves this by allowing practitioners to create and plug in their own analysis and visualization tools, similar to the idea of Probes. This approach ensures that the SST is not tied to a specific method but supports various data exploration techniques. Users can connect or develop their preferred tools for querying, visualization, or advanced analytics, making the SST adaptable to different workflows and project requirements.

Implementation

At the current stage of implementation, any tool compatible with the Neo4j database can be used for querying and exploring the graph data. This enables users to leverage existing graph analysis tools and visualization platforms to gain insights into software maintenance processes. For example, a Tableau BI tool was successfully connected.

If needed in the future, a server-based solution could be implemented to provide customized access to the data, depending on specific user needs. This could include REST APIs, GraphQL, web-based dashboards, or other interfaces tailored to different tasks. The goal of this decision is to ensure flexibility and adaptability, allowing various analysis approaches to integrate seamlessly with the SST system.

4.3 Conclusion

This chapter detailed the design and implementation of the SST system, focusing on its role in integrating heterogeneous maintenance data into a consistent, centralized structure. The design was driven by the key challenges identified in Chapter 2.

- **C1:** Probes and the access layer ensure compatibility with a wide range of tools.
- **C2:** The type system and graph representation formalize and abstract maintenance data.
- **C3:** Graph merging and validation mechanisms ensure consistent and trustworthy integration.

These components collectively allow the SST system to act as a reliable foundation for software maintenance analysis across diverse technologies.

Chapter 5

Evaluation

This chapter evaluates the SST system by examining its usability in real-world software maintenance scenarios. The evaluation focuses on validating the system's ability to integrate heterogeneous data sources, ensure consistency, support graph-based merging, and enable flexible analysis.

5.1 Use case

To evaluate the proposed approach, it is necessary to choose a realistic maintenance scenario that reflects common challenges in software projects. For the main case study, a performance optimization scenario was considered. Imagine a common problem: having a software system that faces performance troubles. As more features are added and more contributors modify the codebase, the system becomes more complex, and more issues may arise. In this case study, the SST system is used to help analyze the project, detect potential issues, and integrate various maintenance tools to improve performance.

A good evaluation case should involve a solid web project that has been actively developed, has multiple contributors, and requires continuous improvements. For this reason, the *Spring Petclinic* project was selected as the main project for the case study. Spring Petclinic is a well-known open-source application that serves as a reference project within the Spring ecosystem. It has been under development since 2013, making it a suitable candidate for maintenance evaluation. The project simulates a management system for a veterinary clinic, allowing users to manage pet owners, register pets, document visits, and view veterinarians.

Spring Petclinic is a good candidate for evaluation for several reasons. Firstly, it is an open-source project, meaning that all details about its code and development history are fully accessible. Secondly, the project has been actively developed for more than a decade, providing a realistic example of a long-term software system that requires ongoing maintenance. Thirdly, the project is built using Java and the *Spring Boot framework*, one of the most popular technology stacks for web development today [16]. This makes the evaluation more relevant to modern software engineering practices.

By applying the SST framework to the Petclinic project, the evaluation demonstrates how the approach can support software maintenance tasks in practical scenarios. The results provide insights into the effectiveness of the system in handling heterogeneous maintenance data and integrating multiple tools in a centralized way.

5.1.1 Probes

To analyze the performance issues in the case study, we might need different types of data. First, we need performance measurements to identify bottlenecks and understand which parts of the system are slow. Second, we need the full structure of the code, including its classes, methods, and dependencies. This helps us see how the problematic areas fit into the overall system and how performance issues spread across components. Finally, we need information about developer contributions, allowing us to trace performance problems back to specific changes and understand which developers are most familiar with the affected code. These three types of data are provided by separate probes, following the probe abstraction model described in Section 4.2.1. By combining these three perspectives, we can gain a clear understanding of the problem and make informed maintenance decisions (Figure 5.1).

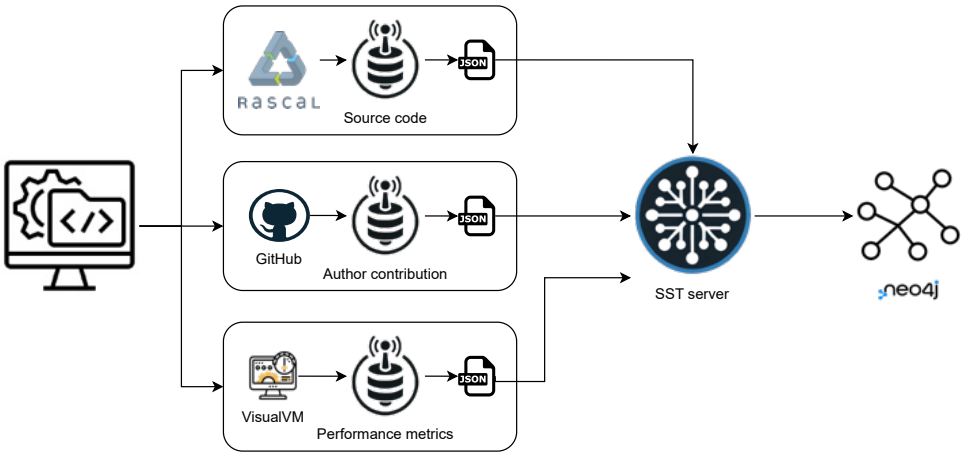

Figure 5.1: Types used in the performance probe.

Performance

The performance probe is implemented using the *VisualVM* analysis tool. It provides runtime performance measurements by tracking method execution time during runtime. This allows us to identify slow parts of the system and understand where optimizations are needed. The probe captures various runtime metrics and integrates them into the SST to provide a detailed view of performance issues.

The data collected by the probe includes method execution times, CPU usage, number of invocations, I/O operation time, and time spent in outgoing method calls. Each performance measurement is linked to a specific method in the codebase, creating a structured representation. The types used in this probe are shown in Figure 5.2, and an example graph of the collected data is presented in Figure 5.3.

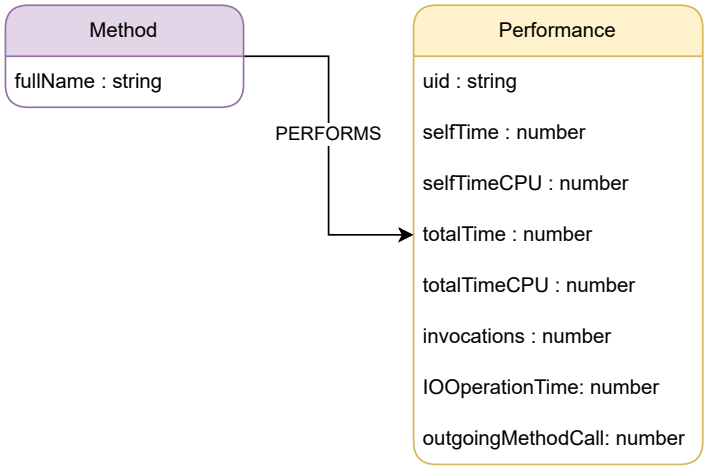

Figure 5.2: Types used in the performance probe.

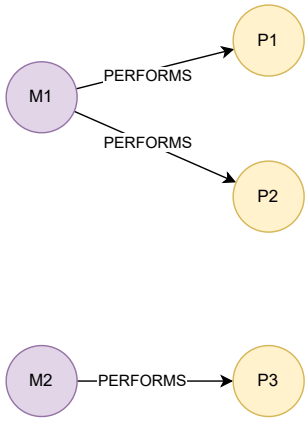

Figure 5.3: Example of performance data in the system.

Code Structure

The code structure probe is implemented using *Rascal*, a domain-specific tool for analyzing software, including Java, C++, Python, and other projects. It extracts detailed information about the static architecture of the software, including its methods, classes, packages, and dependencies. This helps us understand how different parts of the system are organized and how they interact. By integrating this structural information into the system, we can analyze dependencies and locate performance issues within the broader context of the project.

The probe provides data on the hierarchy of the project, including packages containing files, files containing classes, and classes containing methods. Additionally, it tracks method calls, helping to visualize how different functions interact. This structured view of the codebase is crucial for identifying areas that may require refactoring. The types used in this probe are shown in Figure 5.4, and an example of the graph data is presented in Figure 5.5.

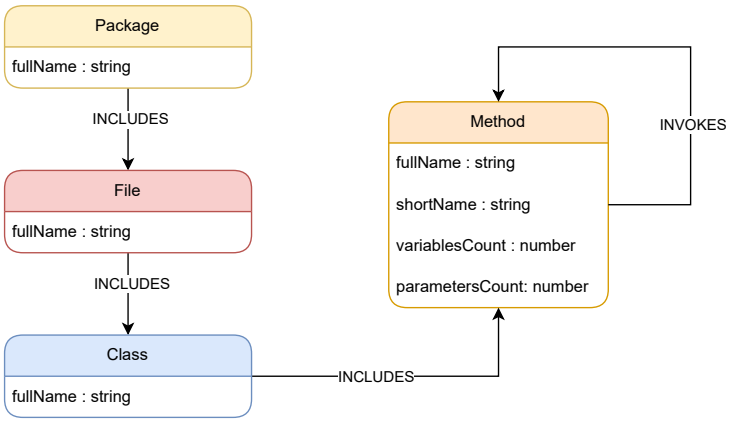

Figure 5.4: Types used in the code structure probe.

Author Contribution

The author contribution probe is implemented using a custom Python script that analyzes Git repository data. It provides insights into how different developers have contributed to the project methods. This information helps in identifying the right people to work on fixing performance problems and understanding how recent modifications may have affected the system.

The probe collects data on developers and their contributions at the method level. Each method is linked to the author who modified it. The types used in this probe are shown in Figure 5.6, and an example of the collected data is presented in Figure 5.7.

5.1.2 Integration

Before pushing any data to the system, each probe must be registered in the SST. This process ensures that all three probes are compatible with each other on the type level. Figure 5.8 shows the resulting global type schema after registering all three probes. This schema serves as a unified schema for the entire process, ensuring

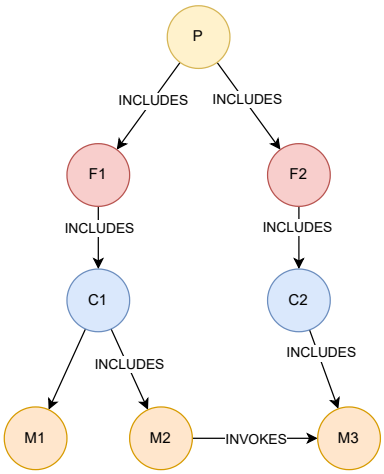

Figure 5.5: Example of code structure data.

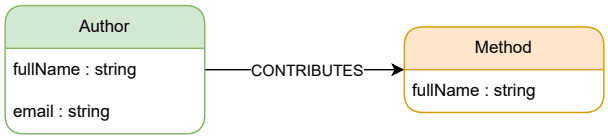

Figure 5.6: Types used in the author contribution probe.

consistency between different probes.

Once the probes are registered, each data graph can be pushed to the SST independently. This results in a unified global graph that represents the system's combined data, linking performance measurements, code structure, and developer contributions in a single graph. The integration process follows the graph merging mechanism described in Section 4.2.4, which ensures that equivalent nodes are consistently merged based on type definitions and equivalence rules. Figure 5.9 illustrates the final merged graph, showing how different nodes are connected.

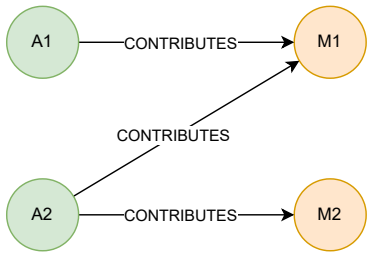

Figure 5.7: Example of author contribution data.

5.1.3 Graph analysis

As a result of merging the data from all three probes, the SST system builds a unified graph that represents the analysed Petclinic project. The final graph includes 360 nodes and 549 edges, combining the structure of the code with performance data and developer contributions. Table 5.1 shows a breakdown of the node and relationship types included in the graph, giving a clear overview of the data.

Type	**Count**
Package	6
File	30
Class	36
Method	182
Author	14
Performance	77
Includes (relationship)	223
Performs (relationship)	78
Invokes (relationship)	142
Contributes (relationship)	106

Table 5.1: Node and relationship counts in the Petclinic project

The Neo4j database allows us to perform various types of queries on the project

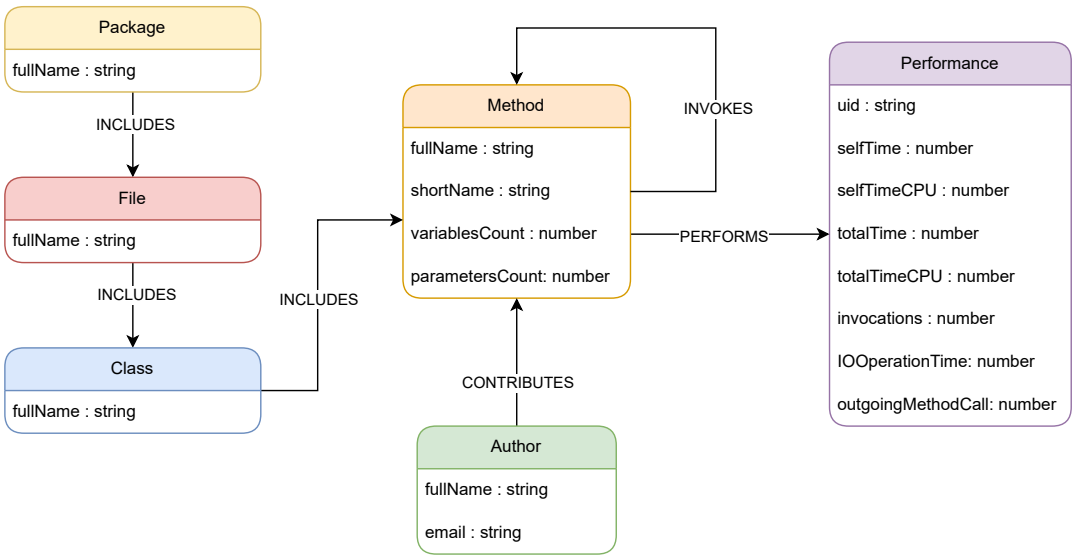

Figure 5.8: Global type schema after probe registration.

graph to analyze different aspects of the project and the software maintenance process. This is made possible by the flexible data access layer described in Section 4.2.5, which enables external tools and users to interact with the integrated graph through standard query interfaces. This section demonstrates how we can use Cypher queries to explore performance issues, dependencies, and contributors related to slow methods.

To begin, we need to identify the slowest methods in the system based on execution time. The following query retrieves the slowest methods along with their corresponding class, file, and package, providing an overview of their location in the project.

Listing 5.1: Query to find slowest methods

```
MATCH (m:Method)<-[:INCLUDES]-(c:Class)
<-[:INCLUDES]-(f:File)<-[:INCLUDES]-(p:Package)
```

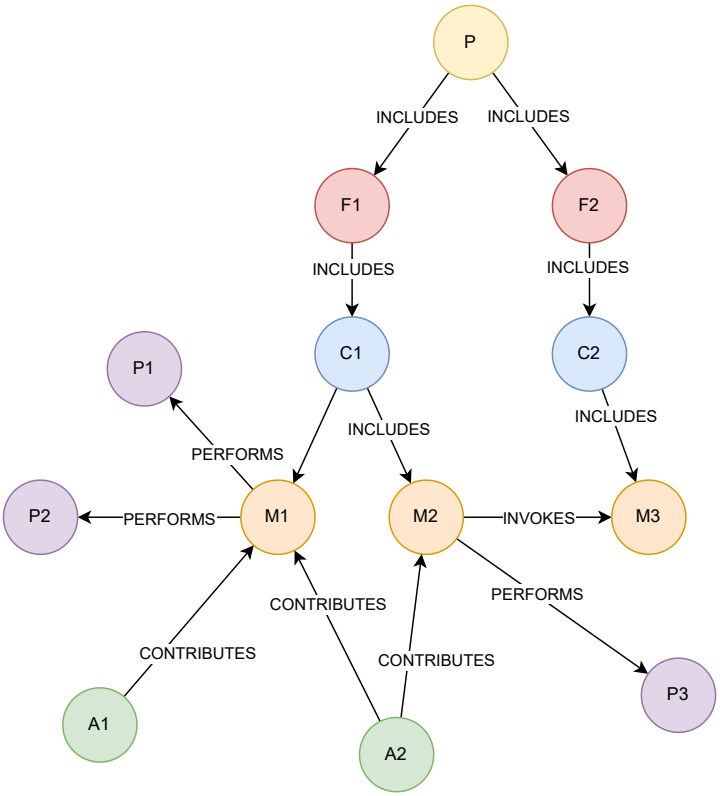

Figure 5.9: Merged global graph representing the integrated system data.

```
MATCH (m)-[:PERFORMS]->(perf:Performance)
WITH  m, c, f, p, perf
ORDER BY perf.TotalTime DESC
LIMIT 5
RETURN m, c, f, p, perf
```

The result of this query is shown in Figure 5.10, displaying the most time-consuming methods in the project along with their locations.

Once we have identified the slowest methods, the next step is to analyze which other methods invoke them. This helps us understand the potential impact and

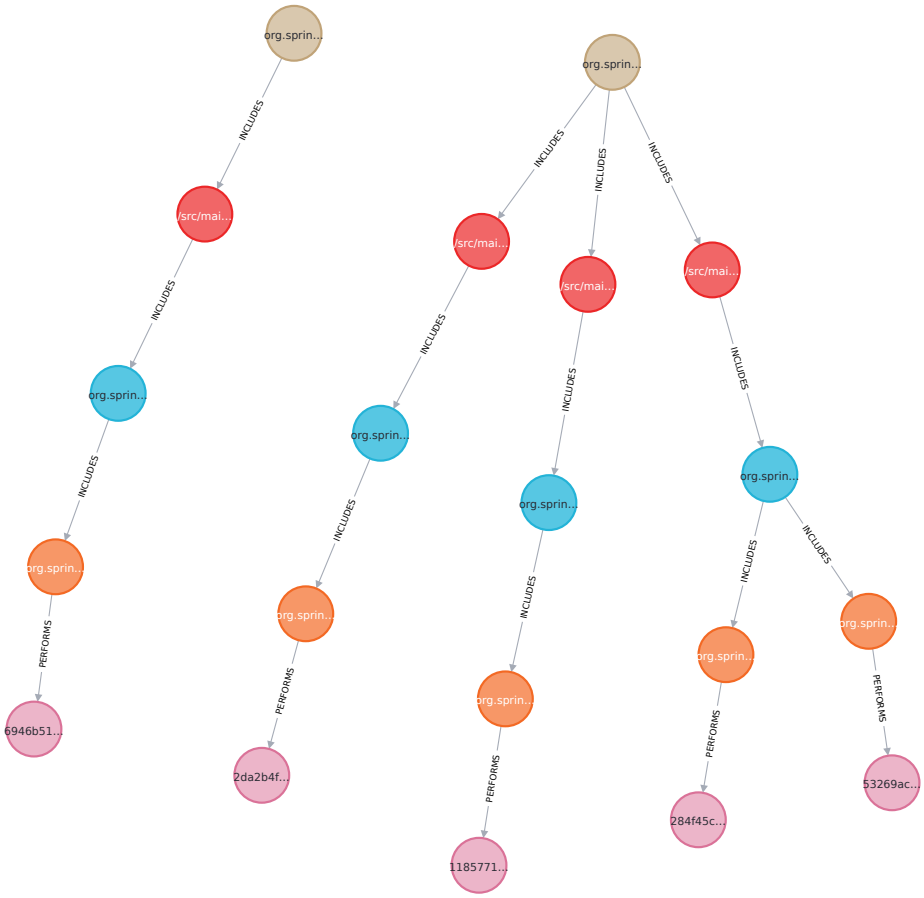

Figure 5.10: Slowest methods and their locations.

dependency of these slow methods on the overall system performance.

Listing 5.2: Query to find methods invoking slow methods

```
MATCH (caller:Method)-[:INVOKES]->(slow:Method)-[:PERFORMS]->(perf:
    Performance)
WITH slow, perf, caller
ORDER BY perf.TotalTime DESC
LIMIT 10
RETURN caller, slow, perf
```

Figure 5.11 presents the results, showing the connections between the slowest

methods and their callers.

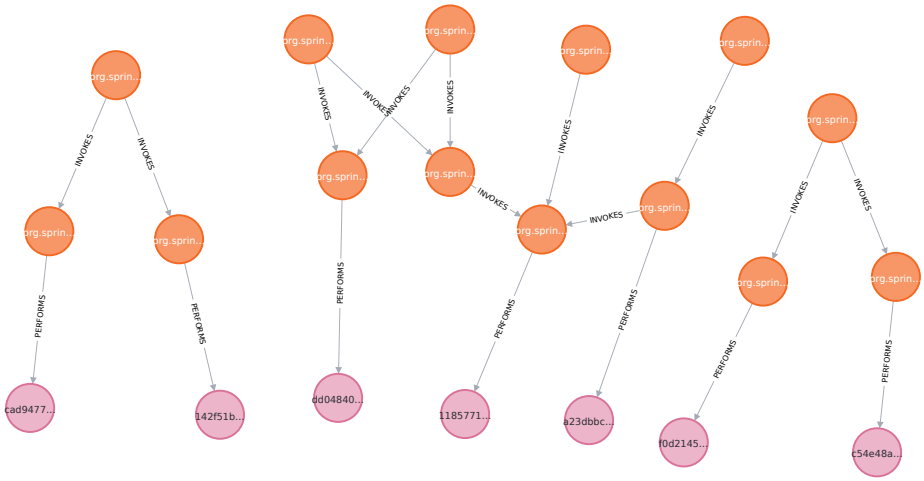

Figure 5.11: Methods invoking slow methods.

To address performance issues, it is important to identify the developers who contributed to the affected methods. This helps determine who is most familiar with the code and can work on refactoring and optimizing it.

Listing 5.3: Query to find contributors of slow methods

```
MATCH (a:Author)-[:CONTRIBUTES]->(slow:Method)-[:PERFORMS]->(perf:
    Performance)
MATCH (c:Class)-[:INCLUDES]->(slow)
WITH a, slow, perf, c
ORDER BY perf.TotalTime DESC
LIMIT 10
RETURN a, slow, c
```

The output of this query, visualized in Figure 5.12, highlights the contributors who have modified the slowest methods, assisting in assigning the necessary maintenance decisions.

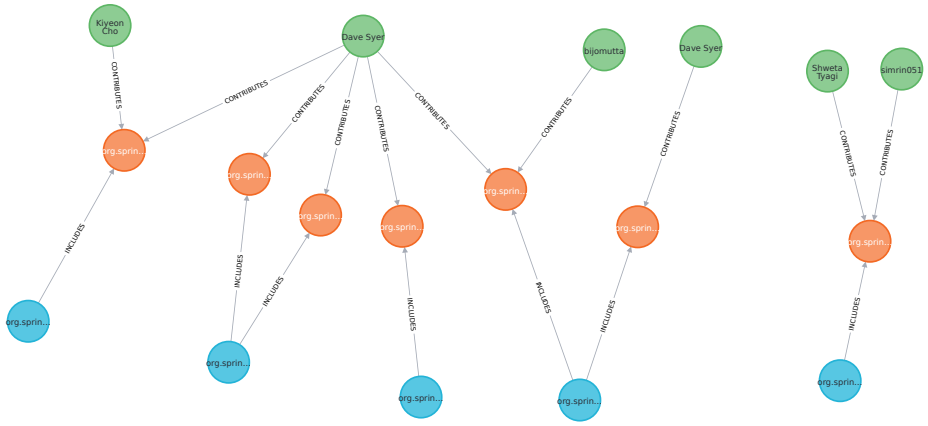

Figure 5.12: Contributors of slow methods, responsible for potential optimizations.

These queries illustrate how Neo4j can be utilized to analyze the project graph, identify performance bottlenecks, assess their impact, and find the right contributors to address the issues. As a result of using the SST and leveraging these graph analysis queries, maintenance tasks can be better prioritized and managed.

5.1.4 Data visualization

Another SST data usage scenario includes data representation and visualization. After executing all three probes, the system representation was stored as a graph in the Neo4j database. To analyze this data effectively, we connected the BI tool Tableau to design interactive visualizations and graphs, making it easier to understand the project data.

To gain insights into the distribution of code modifications, we created a visualization that shows the contributions of each author, measured by the number of lines of code edited. As seen in Figure 5.13, this chart helps identify the main contributors to the project. It highlights the presence of a dominant author who has made significantly more changes than others. Understanding these contributions is essential

when identifying responsible developers for maintaining or optimizing specific parts of the codebase.

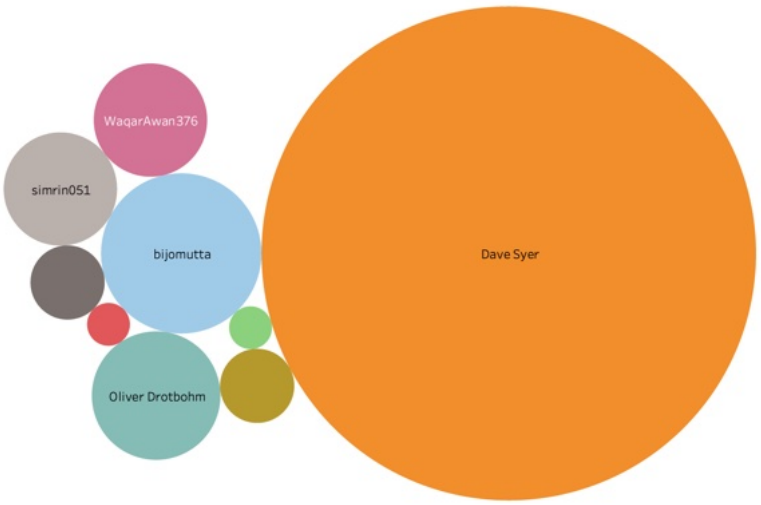

Figure 5.13: Visualization of author contributions based on lines of code edited.

To analyze performance issues, we created a bar chart of the execution times of methods in the system. Figure 5.14 presents a breakdown of method execution time based on three different metrics. This visualization enables a deeper understanding of how methods contribute to performance issues and where optimizations should be prioritized.

Since resolving performance issues requires knowledge of both slow methods and their contributors, we created a chart (Figure 5.15) that links authors to the slow methods. This chart provides an intuitive way to identify the right developer to contact when optimizing specific slow methods.

These visualizations illustrate how the framework allows us to explore project data efficiently. For the use case scenario, Tableau serves as a powerful tool for

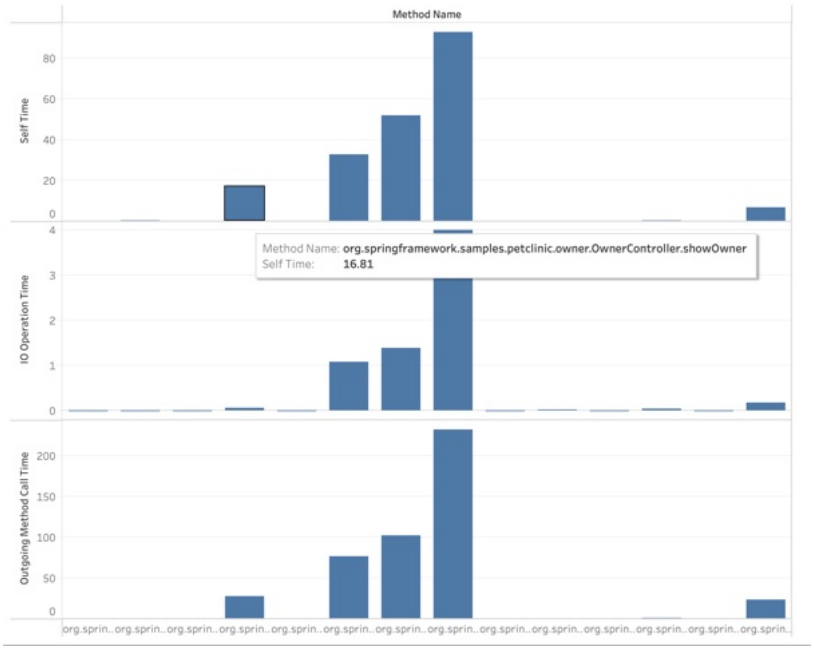

Figure 5.14: Execution time of methods categorized by self-time, I/O operations, and outgoing calls.

transforming complex graph-based information into actionable insights.

5.2 Microservices Use Case

The SST system was also used in research MEng project that focused on improving software maintenance in Java-based microservices systems. This research explored how team collaboration, project structure, and service documentation could be improved using graph-based analysis. To support this, the SST system was selected as the foundation for data integration and storage. Its ability to unify heterogeneous data and provide a structured format made it a strong fit for this use case.

In this project, several probes were created to collect and organize relevant maintenance and development data. Each probe targeted a specific aspect of the codebase

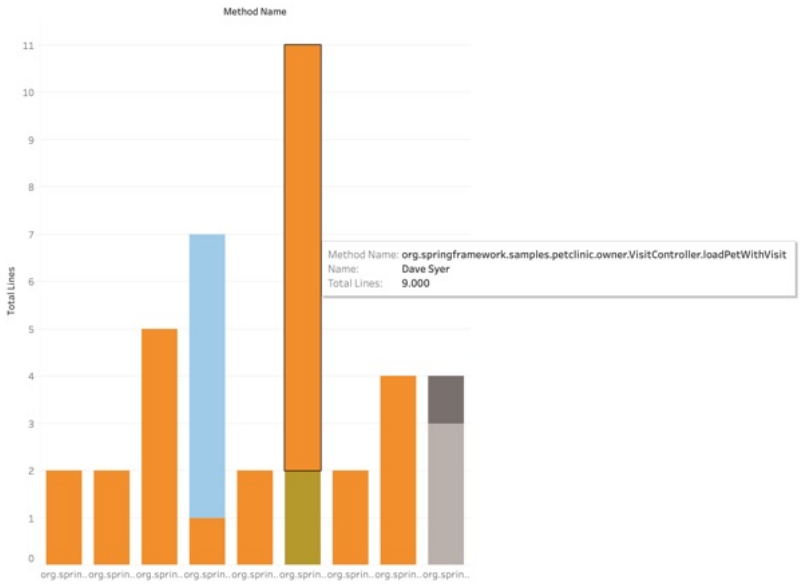

Figure 5.15: Contributions of authors to slow methods, helping identify the best candidates for optimization tasks.

or team activity. Below is a brief overview of the probes used:

1. **Most Recent Contributor:** Identifies the last developer who modified a method. This helps in resolving bugs by involving the person most familiar with recent changes.
2. **List of Contributors:** Gathers all developers who contributed to a specific method or class. Useful for documentation and communication.
3. **Top Contributor:** Highlights the developer with the most contributions to a method, helping identify domain experts.
4. **File Contributors:** Lists all developers who worked on a file. Helps in clarifying responsibilities.
5. **Author Relation:** Measures collaboration strength between developers based on joint contributions. Supports team evaluation and collaboration insights.

6. **Endpoints:** Extracts all REST API endpoints from the system. Useful for identifying project API structure.
7. **Bean Data:** Collects all registered Spring beans and their dependencies. Helps identify configuration issues and understand system structure.
8. **Dependencies List:** Extracts library and framework dependencies across microservices. Helps track outdated or vulnerable dependencies and supports maintenance tasks.

After all the probes registered and pushed their data to the SST, the system merged this information into a single, consistent graph (Figure 5.16). This unified graph allowed practitioners to explore relationships between contributors, source code elements, and service components all in one place.

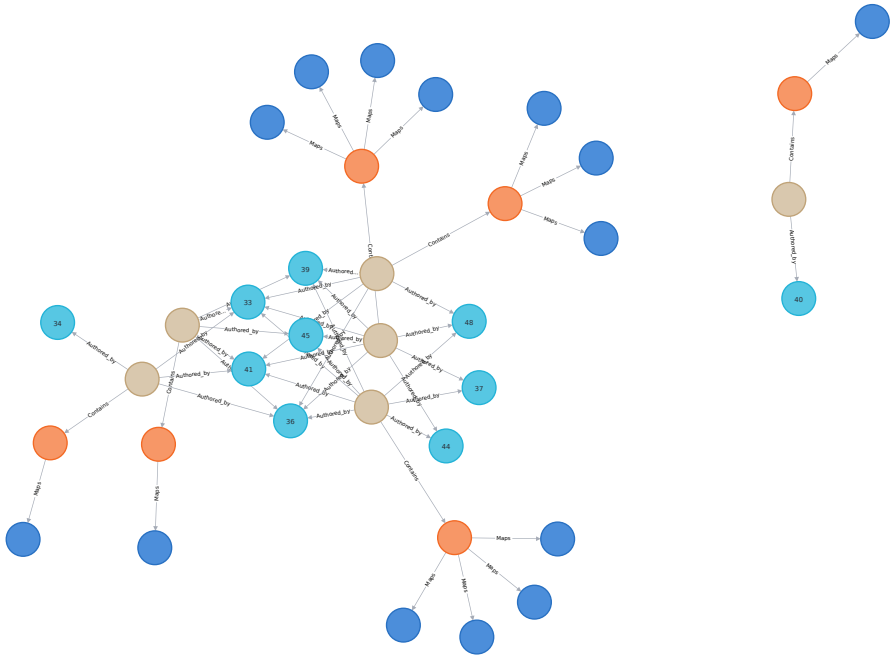

Figure 5.16: Part of the final merged graph.

This case study shows that the SST system works well for different types of maintenance tasks and data structures. It is especially useful in microservice architectures, where data comes from many sources and services. In addition, the example demonstrates that the system is suitable and useful for external scenarios, having a fast learning curve that makes it easy to get started with the system.

5.3 Large System Use Case

Another research group at Université Côte d'Azur has shown interest in using the SST system to support their work on understanding variability in large software systems. Their study, presented by Mortara et al. and titled *Visualization of Object-Oriented Variability Implementations as Cities* [15], introduces a tool called *VariCity*.

VariCity visualizes object-oriented software systems as 3D cities—where classes are shown as buildings and relationships such as inheritance or usage are represented as streets. This approach helps developers explore and understand how variability is implemented across a system, particularly in projects that rely on inheritance, method overloading, or design patterns.

For this use case, VariCity was used to visualize variability in large Java-based software systems using pre-generated data produced by the Symfinder toolchain [14]. Symfinder analyzes codebases to detect variability implementations, and exports this information in structured JSON files. These files describe class-level relationships, such as inheritance and usage, which are then used by VariCity to construct interactive 3D visualizations. The evaluation was performed using data from ten large open-source projects, including Apache Maven, NetBeans, JUnit, and others.

The SST system can support this research by addressing common data integration

and consistency challenges in large-scale systems:

- **Structured storage:** SST allows variability data to be stored in a formal, unified graph format that includes classes, interfaces, methods, and their relationships.
- **System-wide consistency:** By merging data project subgraphs, SST ensures that the integrated information remains coherent and accurate.
- **Single data provider:** SST can act as a data provider for tools like VariCity, making it easier to generate consistent and reliable visualizations.

The SST system was able to successfully integrate and manage all data graphs produced by the Symfinder toolchain across the evaluated projects. In the largest scenario with the Apache NetBeans project, the SST handled more than 30,000 nodes and more that 30,000 relationships without significant integration or performance issues (Table 5.2). This demonstrates the system's scalability and its ability to work effectively with large and complex software systems, maintaining consistent structure and performance throughout the integration process.

Type	**Count**
Class	2,846
Method	29,535
Interface	674
Implements (relationship)	933
HasMethod (relationship)	29,537

Table 5.2: Node and relationship counts in the NetBeans dataset

This use case highlights two key strengths of the SST system:

1. **External collaboration:** The interest from another research group confirms that SST is applicable beyond its initial scope and can support a range of software engineering studies.

2. **Scalability and consistency:** The ability to manage large-scale, structured data makes SST well-suited for visualization and analysis in complex systems and scenarios.

5.4 Limitations

While the SST ensures data consistency and integration, it does not guarantee the correctness of the data provided by individual probes. Although validation and integration mechanisms can detect inconsistencies and highlight potential issues, the system does not actively verify or correct the accuracy of the input data itself.

Another key limitation of the approach is the requirement for users to develop and configure probes. This process involves structuring the data in the required format, defining types, and specifying node equivalence rules. In some cases, defining equivalence can be challenging, especially for data that lacks natural unique identifiers, such as performance measurements.

Additionally, the reliance on graph-based storage, while beneficial for managing relationships and dependencies, may not be suitable for all types of maintenance data or specific use cases. These limitations highlight areas where further improvements and adaptations could enhance usability of the system.

Chapter 6

Conclusion

6.1 Results

The main goal of the work is to explore the application of the Single Source of Truth approach in the software maintenance domain. The work focuses on designing and implementing a system that enables structured integration of heterogeneous maintenance data, ensuring consistency and reliability.

The thesis investigates key design decisions necessary for such implementation, including data abstraction and schema, integration strategies, data formats and other. The proposed solution introduces, implements and validates a way to collect, merge, and analyze maintenance data from multiple sources.

To validate the proposed approach, the SST system was applied to an open-source software project, demonstrating its ability to support real-world maintenance tasks such as performance analysis, code structure exploration, and developer contribution tracking. The results show that the SST concept effectively addresses the challenges of data integration in software maintenance, providing a reliable foundation for analysis

and informed decision-making. In addition to this case study, the SST was also successfully used in a microservices-based project and in a large-scale system variability analysis.

6.2 Discussion

This section discusses several open questions that arise from the results of the SST system and help to clarify possible future directions.

What maintenance scenarios are most suitable for SST? Even though the SST system is built as a flexible and general solution, we can highlight several types of maintenance scenarios and projects where its usage is especially reasonable. These include legacy systems with a long development history, missing documentation, or overcomplicated structure. In such cases, information recovery becomes important, and maintenance decisions require structured and trustworthy data. SST is particularly useful when critical decisions need to be made based on fragmented or outdated information, providing a consistent and unified view of the system's state.

What data is actually needed for SST to operate effectively? While SST can support various types of maintenance data, to build a consistent and connected knowledge graph it requires some kind of skeleton. According to our usage, a form of code structure—such as files, classes, and methods—is essential and serves as the base layer. Other data sources—such as performance metrics, authorship information, security issues, or dependency data—are layered on top of this structure. In more advanced cases, it can also be beneficial to include data from ticket tracking systems, documentation, or other sources that reflect the project's maintenance and development history.

Can we validate data completeness using SST? This question opens an interesting direction for future work. One possible approach is based on the probe concept. By comparing the current project setup with similar maintenance scenarios, it becomes possible to identify missing perspectives and suggest existing probes that could be reused. Another approach is based on the global type schema, which defines what types of data are expected for the analysis. By comparing this schema with the actual data provided by the probes, the system can detect gaps in coverage or underdeveloped areas. Together, the schema and probe mechanisms provide a structured way to evaluate and improve the completeness of the collected data.

6.3 Future Work

One of the main future goals of this project is to test the solution in a real maintenance-driven process. So far, the SST has been developed based on theoretical maintenance needs and best practice ideas, but real-world usage may present different challenges and requirements. It is difficult to predict how practitioners will interact with the system, what specific scenarios will appear, and what additional techniques or adjustments will be necessary to support real maintenance workflows.

Another important direction is to improve system performance and extend support for more advanced data usage and analysis scenarios. This includes wider integration with external data access and analysis tools, improved ways to explore and query data, and more flexible usage in large-scale projects. Future work could also explore automated suggestions for missing data and schema validation mechanisms to detect incomplete or inconsistent information. Additionally, it may include the active development of new probes based on common maintenance patterns and recurring tasks.

Having a larger set of reusable probes and maintenance scenarios could eventually lead to the idea of a shared probe marketplace and a community built around the SST system.

Bibliography

[1] G. Buchgeher, R. Weinreich, and H. Huber. A platform for the automated provisioning of architecture information for large-scale service-oriented software systems. In *Software Architecture: 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24–28, 2018, Proceedings 12*, pages 203–218. Springer, 2018.

[2] G. Canfora, M. Di Penta, and L. Cerulo. Achievements and challenges in software reverse engineering. *Communications of the ACM*, 54(4):142–151, 2011.

[3] Z. Chen, B. Pan, and Y. Sun. A survey of software reverse engineering applications. In *Artificial Intelligence and Security: 5th International Conference, ICAIS 2019, New York, NY, USA, July 26–28, 2019, Proceedings, Part IV 5*, pages 235–245. Springer, 2019.

[4] S. E. S. Committee et al. Ieee standard for software maintenance. *IEEE Std*, pages 1219–1998, 1998.

[5] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather. Programl: Graph-based deep learning for program optimization and analysis. *arXiv preprint arXiv:2003.10536*, 2020.

[6] K. Erdil, E. Finn, K. Keating, J. Meattle, S. Park, and D. Yoon. Software maintenance as part of the software life cycle. *Comp180: Software Engineering Project*, 1:1–49, 2003.

[7] G. C. Gannod and B. H. Cheng. A framework for classifying and comparing software reverse engineering and design recovery techniques. In *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*, pages 77–88. IEEE, 1999.

[8] A. Gupta and S. Sharma. Software maintenance: Challenges and issues. *Issues*, 1(1):23–25, 2015.

[9] R. Hai, C. Koutras, C. Quix, and M. Jarke. Data lakes: A survey of functions and systems. *IEEE Transactions on Knowledge and Data Engineering*, 35(12): 12571–12590, 2023.

[10] B. Khan, S. Jan, W. Khan, and M. I. Chughtai. An overview of etl techniques, tools, processes and evaluations in data warehousing. *Journal on Big Data*, 6, 2024.

[11] S. Khanna, A. Shah, S. Jain, and L. Ramanathan. Software maintenance: Challenges and issues and models for reducing the maintenance cost. *International journal of advanced research in computer science*, 8(3), 2017.

[12] V. Lenarduzzi, A. Sillitti, and D. Taibi. A survey on code analysis tools for software maintenance prediction. In *Proceedings of 6th International Conference in Software Engineering for Defence Applications: SEDA 2018 6*, pages 165–175. Springer, 2020.

[13] S. Martínez-Fernández, P. Jovanovic, X. Franch, and A. Jedlitschka. Towards automated data integration in software analytics. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, pages 1–5, 2018.

[14] J. Mortara, X. Tërnava, and P. Collet. symfinder: A toolchain for the identification and visualization of object-oriented variability implementations. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B*, pages 5–8, 2019.

[15] J. Mortara, P. Collet, and A.-M. Dery-Pinna. Visualization of object-oriented variability implementations as cities. In *2021 Working Conference on Software Visualization (VISSOFT)*, pages 76–87. IEEE, 2021.

[16] M. Mythily, A. S. A. Raj, and I. T. Joseph. An analysis of the significance of spring boot in the market. In *2022 international conference on inventive computation technologies (ICICT)*, pages 1277–1281. IEEE, 2022.

[17] R.-H. Pfeiffer and J. Aaen. Tools for monitoring software quality in information systems development and maintenance: five key challenges and a design proposal. *International Journal of Information Systems and Project Management*, 12(1): 19–40, 2024.

[18] M. Queiroz, P. Tallon, and T. Coltman. Data value and the search for a single source of truth: What is it and why does it matter? 2024.

[19] O. Rodriguez-Prieto, A. Mycroft, and F. Ortin. An efficient and scalable platform

for java source code analysis using overlaid graph representations. *IEEE Access*, 8:72239–72260, 2020.

[20] F. Ruiz and J. R. Hilera. Using ontologies in software engineering and technology. In *Ontologies for software engineering and software technology*, pages 49–102. Springer, 2006.

[21] P. Samarasekara, R. Hettiarachchi, et al. A comparative analysis of static and dynamic code analysis techniques. *Authorea Preprints*, 2023.

[22] P. Somasekaram, R. Calinescu, and R. Buyya. High-availability clusters: A taxonomy, survey, and future directions. *Journal of Systems and Software*, 187: 111208, 2022.

[23] E. B. Swanson. The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering*, pages 492–497, 1976.

[24] S. Zappa, C. Franciosi, A. Polenghi, and A. Voisin. Ontology-based digital twin for maintenance decisions in manufacturing systems: an application at laboratory scale. *IFAC-PapersOnLine*, 58(8):13–18, 2024.