Comparative Study of Static and Dynamic Metric Tools for Coupling and Cohesion in Microservices

By:

Adarsh Ajit

ID: 24026492

Supervised By

Professor J.J. Collins

Submitted to the University of Limerick, August 2025 in Partial Fulfillment of the Requirements for the Degree of Master of Science In Software Engineering

Abstract

This dissertation presents a comparative study of two metric-based approaches to evaluating microservice quality, with a focus on coupling and cohesion. As organisations increasingly adopt microservice architectures (MSA) over monolithic designs, validating the quality of service decomposition becomes critical to maintainability and system resilience. This work compares two metric suites, MicroValid a static validation framework, and the runtime metrics proposed by Bogner et al. to assess their consistency and applicability in capturing architectural quality.

To identify relevant quality attributes, a graph was constructed linking common code smells to corresponding software metrics. This visualisation highlighted coupling and cohesion as the most prominent and frequently measured attributes, justifying their selection as the focus of this study. MicroValid assesses microservices quality using coefficient-of-variation-based metrics such as Dependencies Composition, Responsibilities Composition, and Strongly Connected Components (SCC), while Bogner et al.'s approach derives coupling and cohesion metrics from distributed trace data.

This dissertation will develop a test suite to apply both tools, MicroValid and Bogner et al.'s runtime metric suite to the microservice codebases, using both model-based and implementation-based inputs. The goal is to examine whether the insights produced by static and runtime metrics align, and to assess the reliability of MicroValid in reflecting architectural quality as observed in a running system.

Declaration

Title: Comparative Study of Static and Dynamic Metric Tools for Coupling and Cohesion in Microservices

Author: Adarsh Ajit

Award: Master of Science

Supervisor: Professor J.J. Collins

I herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This paper has not previously been presented in identical or similar form to any other Irish or foreign examination board.

Adarsh Ajit

17th August, 2025

Consent of Publication

I hereby consent that this dissertation document can be made publicly available in electronic format for research purposes.

Adarsh Ajit

17th August, 2025

Use of AI/GenAI

I declare that I used Generative AI tools such as ChatGPT and Grammarly, to assist with formatting and language refinement of the dissertation content.

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Professor J.J. Collins, for his invaluable guidance, support, and encouragement throughout the course of this research. His expertise and insights have been instrumental in shaping the direction and outcome of this dissertation.

Finally, I would like to extend my heartfelt thanks to my family and friends for their unwavering support and understanding throughout this journey. Their encouragement and belief in me have been a constant source of motivation.

Adarsh Ajit

17th August 2025

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

This dissertation investigates the relationship between static and dynamic metric-based approaches to assessing the quality of microservice architectures. It first identifies the dominant quality attributes through a mapping study of their association with common code smells. It then conducts a direct empirical comparison of two distinct analysis paradigms: a static tool, MicroValid (Cojocaru et al., 2019), and a dynamic tool from Bogner, Schlinger, Wagner and Zimmermann (2019), which measures runtime behavior. By applying both tools to the same systems, this research examines whether static measurements align with their runtime equivalents.

Microservice architecture has emerged as a dominant paradigm for building scalable and adaptable systems, offering significant advantages in agility, fault isolation, and independent deployment (Dragoni et al., 2017). This architectural style allows large, complex applications to be decomposed into a suite of small, autonomous services that can be developed, deployed, and scaled independently (Fowler and Lewis, 2014). This modularity aligns with modern DevOps practices and enables teams to adapt quickly to changing business requirements, fostering innovation and reducing time-to-market.

Despite these benefits, the distributed and heterogeneous nature of microservices presents significant challenges for quality assessment (Kalske et al., 2017). Unlike monolithic systems with a centralized codebase, the architectural health of a microservice application is defined by the complex, asynchronous interactions between its services. This makes it difficult to apply traditional system-wide metrics, leading to a critical blind spot where architectural decay such as weakening cohesion or tightening coupling can accumulate undetected as a form of architectural technical debt (Bogner, Fritzsch, Wagner and Zimmermann, 2019).

In practice, development teams often lack the tools to monitor these cross-service architectural attributes and consequently fall back on local, code-level

metrics (Bogner, Fritzsch, Wagner and Zimmermann, 2019). Empirical studies have found that industry professionals predominantly rely on tools to assess local metrics like test coverage and code smells, while neglecting architectural-level metrics that assess service interaction and modularity (Bogner, Fritzsch, Wagner and Zimmermann, 2019). This leads to an overreliance on local metrics, which may not reflect the broader quality concerns relevant to microservice architecture, forcing architects to operate on assumption rather than data.

To guide this study, it was noted that both industry and academic research identify coupling and cohesion as the two most critical quality attributes for the maintainability of service-based systems (Bogner, Fritzsch, Wagner and Zimmermann, 2019). These concepts, foundational to structured design for decades (Yourdon and Constantine, 1979), are fundamental goals for achieving a resilient and evolvable microservice architecture (Cojocaru et al., 2019). To confirm this focus, a preliminary analysis was conducted for this dissertation by mapping common code smells to software metrics, which revealed a clear clustering of smells around these two attributes.

To investigate these attributes, this research employs two complementary metric suites. For the static perspective, the study uses MicroValid, a framework that assesses a microservice decomposition from its codebase or design specification, using the coefficient of variation to measure the balance of attributes like coupling and cohesion (Cojocaru et al., 2019). For the dynamic perspective, the study employs the runtime analysis tool proposed by Bogner, Schlinger, Wagner and Zimmermann (2019), which leverages distributed tracing data from systems like Zipkin to extract metrics from the executing system.

The core of this dissertation involved applying both tools to four real-world, open-source microservice applications to collect paired metric sets. For each system, static metrics were generated from a reverse-engineered architectural model (JSON input), while dynamic metrics were collected from a running instance instrumented with Zipkin. The key finding from this comparative analysis reveals that while both paradigms effectively identify architectural coupling issues, static analysis of cohesion proves to be an unreliable predictor of runtime behavior, with the two approaches often yielding contradictory conclusions.

1.1 Research Questions

This dissertation is guided by the following research questions:

RQ1: Which software quality attributes are predominantly associated with code smells in microservice architectures?

RQ2: How do MicroValid static coupling and cohesion metrics compare to Bogner et al., runtime metrics when applied to the same microservice systems?

1.2 Research Contribution

This dissertation makes the following contributions:

- It present's the first direct empirical comparison of MicroValid's static analysis framework and Bogner et al.'s runtime analyis tool. By applying both to real-world microservice systems, this study establishes the statistical correlation between static and runtime measurements of coupling and cohesion.
- The research provides architects with clear, evidence-based guidance on the predictive power and limitations of static analysis. It identifies the specific conditions under which static coupling and cohesion metrics can serve as reliable proxies for runtime quality, enabling teams to make earlier, more informed design decisions and mitigate architectural decay.
- This thesis contributes a foundational mapping of common microservice code smells to underlying software metrics from academic literature. This synthesis empirically justifies the study's focus by demonstrating that coupling and cohesion are the dominant quality attributes for assessing architectural health.
- To ensure transparency and foster future research, this work delivers a complete replication package. The package includes analysis scripts, configuration files, and processed data from the case studies, providing a robust foundation for researchers to validate these findings and conduct further comparative studies of microservice analysis tools.

1.3 Methodology

The following steps were taken to accomplish the dissertation:

- Conducting a comprehensive literature review to survey related work on software metrics, microservice architecture, and existing static and dynamic analysis tools.
- Selecting MicroValid as the static analysis framework and Bogner et al.'s runtime metric tool as the dynamic analysis tool for evaluation.

- Deploying the selected microservice systems in a test environment with distributed tracing enabled using Zipkin.
- Installing and configuring the MicroValid framework and the runtime metrics tool to analyze the chosen systems.
- Collecting the corresponding static and runtime metrics on coupling and cohesion for each system, ensuring the measurements are comparable.
- Performing statistical analysis on the paired metric sets, including the computation of correlation coefficients between the static and runtime values.
- Evaluating the results to assess how well MicroValid's static scores reflect the runtime quality of the system and discussing the reasons for any discrepancies.

1.4 Thesis Outline

The remaining chapters of this dissertation are as follows:

Chapter Two provides a comprehensive literature review, surveying related work on software metrics, microservice architecture, coupling and cohesion, code smells, and existing static and dynamic analysis tools, including MicroValid and Bogner's suite. This establishes the theoretical foundations and contextualizes the research.

Chapter Three details the research methodology, describing the research design and experimental setup. It outlines the microservice systems used as case studies, the installation and configuration of MicroValid and the runtime tracing tool, and the data collection process.

Chapter Four presents the empirical findings from applying both metric tools to the case studies. This chapter compares the static and runtime metric outputs, reports statistical correlations, and analyzes individual service-level results.

Chapter Five concludes the dissertation by interpreting the results in light of the research questions and discussing the implications and limitations of the study. Finally, it summarizes the key contributions and suggests directions for future research.

1.5 Motivation

This research is motivated by the gap between static views of a system's structure and its behaviour observed at runtime. Static tools such as MicroValid (Cojocaru

et al., 2019) provide measurements of coupling and cohesion directly from the codebase, but cannot capture the interaction patterns that emerge during execution. Conversely, dynamic approaches such as the framework proposed by Bogner, Schlinger, Wagner and Zimmermann (2019) derive these metrics from runtime traces, offering insight into the architecture as it actually operates. Understanding how these two perspectives align or diverge is essential for evaluating whether static measurements can serve as reliable indicators of the real-world modularity and maintainability of microservice systems.

Another key motivation lies in the lack of empirical studies that directly compare static and dynamic metrics on the same microservice systems. Although both types of tools have been discussed extensively in academic literature, no existing research has systematically quantified their alignment or divergence with respect to architectural quality, particularly for coupling and cohesion, which are widely recognized as the most foundational indicators of service quality.

By focusing on these two attributes, this study seeks to formally assess the predictive power of static metrics and determine whether they can be reliably used as proxies for runtime quality. The outcome aims to provide practical, evidence-based guidance for architects and development teams, helping them make more informed design decisions, detect architectural degradation earlier, and ultimately build more robust microservice systems.

Chapter 2

Literature Review

This chapter presents an overview of software metrics, going through their classification into product, process, and project metrics, while highlighting their role in assessing software quality. It also covers the fundamentals of microservice architecture and its advantages over traditional monolithic systems. This chapter further explores the challenges of measuring microservices, emphasizing the limitations of conventional metrics in distributed environments. It also examines existing metrics frameworks and standards, along with tools and techniques for identifying code smells in microservices. Finally, it briefly discusses porting techniques from monolithic systems and outlines key research gaps and opportunities for future work.

2.1 Software Metrics Fundamentals

2.1.1 Overview of Software Metrics

Software metrics are quantitative measures applied to all aspects of a software system, such as code, design, processes, project management, etc. IEEE Standards Association (1998) defines software metrics as "a function whose input is software data and whose output is a single numerical value that reflects the degree to which a given attribute affects software quality". These measures convert qualitative software characteristics into quantitative values that inform design choices, quality decisions, and management responses during the course of the software life cycle (Honglei et al., 2009). They help measure design complexity early in the development process, can highlight potential architectural risks, while metrics related to defects and code quality during testing help determine whether the software is ready for release or requires further maintenance (Honglei et al., 2009).

Saini et al. (2014) suggest that software measurement helps to solve issues like design quality evaluation, code complexity assessment, and calculating the amount of development work needed. In a similar fashion, Singh et al. (2011) highlights that informed decisions can be made due to the attributes of the system, which merits capture and helps identify risks, validate designs, and manage test assignments, thereby maximizing resources. All these advantages demonstrate the relevance of software metrics in steering the entire development life cycle from planning to maintenance.

Software measurement emerged in the 1970s, when researchers like Halstead (1977) and McCabe (1976) introduced early methods for quantifying code complexity and control structure. Halstead's software science concentrated on operators and operand counts, whereas McCabe created a metric for control flow complexity called cyclomatic complexity. Albrecht (1979) proposed the Function Points to estimate the size of applications based on their functional requirements. Other advances included the CK metric suite which Chidamber and Kemerer (1994) created as a collection of object-oriented metrics derived from class inheritance trees. Brito e Abreu and Melo (1996) introduced MOOD metrics which emphasized the encapsulation and coupling of object-oriented design features.

2.1.2 Software Metrics Taxonomy

Software metrics are generally grouped into three broad categories: product metrics, process metrics, and project metrics (Honglei et al., 2009). Each category focuses on different aspects of the software lifecycle and serves distinct purposes in quality assessment and project management.

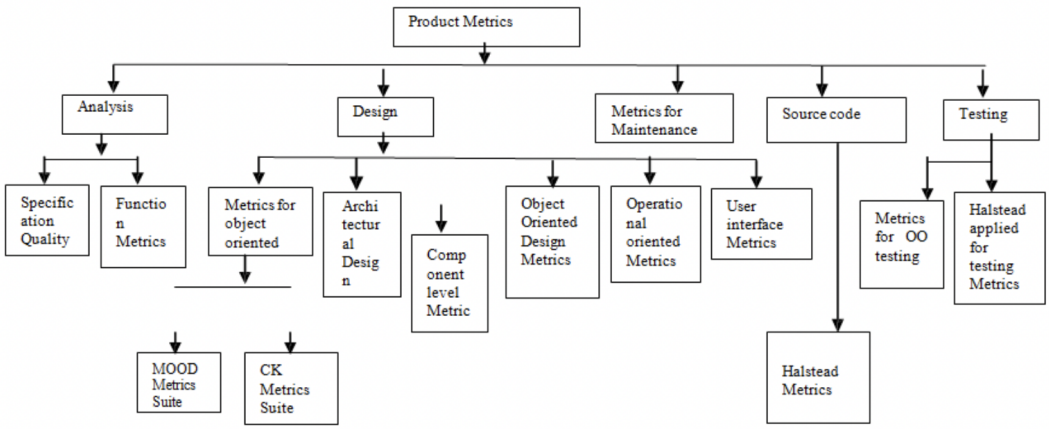


Figure 2.1: Classification of Product Metrics (Fenton and Pfleeger, 1997)

Process Metrics

Process metrics are used to assess the efficiency and quality of the procedures followed during software development. As described by (Honglei et al., 2009), these metrics focus on the duration, cost, and efficiency of the development processes, alongside their improvement capabilities and overall developmental maturity. Typical indicators include lifecycle coverage, defect ratio, and product to process output ratio. These metrics assist in evaluating whether the development processes implemented are in accordance with the industry benchmarks and the project objectives while aiding in predicting the future development directions. As stated by Singh et al. (2011), these process metrics include cost metrics, effort metrics, and even reuse metrics, which aid in monitoring the project and optimising the processes. Saini et al. (2014) highlight that such metrics enable monitoring of the development life cycle and are particularly valuable to senior management who need insight into processes to enable performance assessment and identify areas in need of improvement.

Project Metrics

Project metrics are used to keep track of and monitor specific software projects. These metrics span a wide array of attributes including project size, cost, workload, risk, production efficiency, and even stakeholder satisfaction (Honglei et al., 2009). Furthermore, they are often used at the project level to evaluate ongoing work and make real-time corrections to prevent delays and inefficient use of resources. Singh et al. (2011) defines project metrics as measurement tools focused on management level variables such as team productivity, budget spending, and timelines for deliveries. As Saini et al. (2014) notes, project metrics are useful in pinpointing potential risks and in devising planning enhancements, thus improving product outcomes through refined management techniques. Properly utilizing project metrics leads to improved coordination, insight into the development activities, and evaluation of the project's success.

Product Metrics

Product metrics help assess the internal attributes of software and its quality attributes. As mentioned by Honglei et al. (2009), these metrics evaluate reliability and maintainability during all stages of the software lifecycle: from requirements elicitation through to maintenance after release. Saini et al. (2014) also noted that the type of product metrics used is often based on the programming paradigm used: procedural metrics are more focused on functions and their relationships, while object-oriented metrics tend to focus on classes and structures at class level.

Lines of Code (LOC), Cyclomatic Complexity (McCabe, 1976) and object-oriented metrics from the CK metric suite (Chidamber and Kemerer, 1994), such as Coupling Between Objects (CBO) which measures dependence between classes, and Weighted Methods per Class (WMC), which measures complexity of a class are some of the common product metrics. Singh et al. (2011) further emphasise that product metrics play an important role in assessing system component quality and facilitating comparisons across different software systems.

2.2 Microservices

2.2.1 Microservice Architecture Overview

Microservices have become a widely adopted architectural approach in modern software development, offering an alternative to the traditional monolithic model. Instead of building an entire application as a single, tightly coupled unit, microservices break the system into a suite of small, independent, deployable services each running in its own process and communicating through lightweight mechanisms such as an HTTP resource API (Fowler and Lewis, 2014).

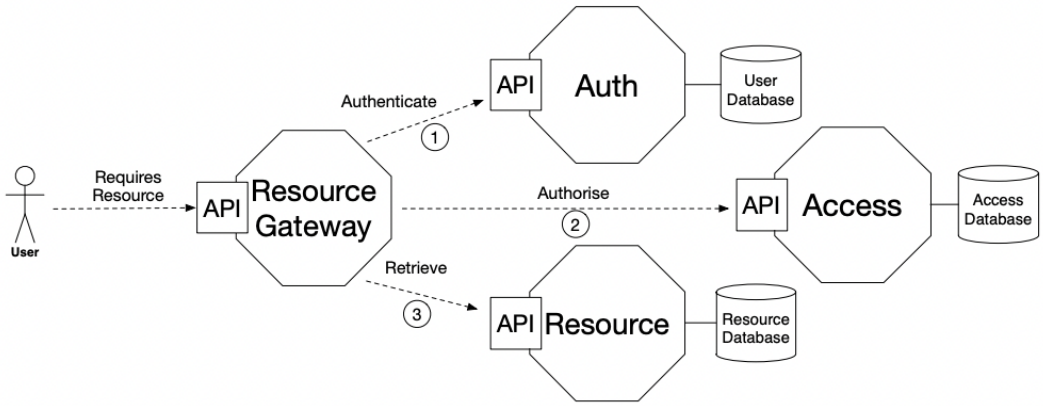


Figure 2.2: Example of Microservice Architecture (Dragoni et al., 2017)

The popularity of microservices indicate a broader industry shift toward modular and distributed systems. This architectural style aligns with concepts like bounded contexts, decentralised data ownership, and continuous delivery. As Fowler and Lewis (2014) points out, the defining feature of microservices is not their size, but their autonomy and their focus on delivering specific business value. These qualities support greater agility, fault tolerance, and scalability particularly for large systems that need to adapt quickly to changing requirements.

Dragoni et al. (2017) trace the evolution of microservices from earlier component-based and service-oriented architectures (SOA). They describe microservices as an

extension of SOA that simplifies service implementation and emphasises lightweight communication and high cohesion within services. Unlike traditional SOA, which often relies on heavyweight enterprise service buses (ESBs), microservices favour decentralised governance and aim for technological heterogeneity, enabling each service to be developed in the most appropriate language or framework. Dragoni et al. (2017) also highlight that microservices promote evolutionary design, where systems grow organically through incremental changes rather than large-scale re-engineering.

Newman (2019) presents a pragmatic perspective, focusing on how organisations can transition from monolithic to microservice architectures through evolutionary patterns. He notes that while microservices can bring significant benefits such as increased scalability, faster release cycles, and improved fault isolation, they also introduce new complexities. These include distributed system challenges such as inter-service communication, service discovery, configuration management, and monitoring. Newman (2019) argues that teams must carefully consider the trade-offs and adopt gradual migration strategies, often starting with the extraction of well-defined components from the monolith.

In summary, microservice architecture represents a significant departure from traditional software design by favouring small, independently deployable services over large, unified codebases. While it offers clear benefits in terms of flexibility, scalability, and team autonomy, it also requires robust infrastructure, automation, and operational maturity. As software systems continue to grow in complexity and scale, microservices offer a promising path, but not without challenges in achieving maintainable and adaptable architectures.

2.2.2 Challenges in Measuring Microservices

Measuring software quality in microservice-based systems presents significant challenges, especially when transitioning from a monolithic architecture. While microservices offer benefits like scalability, independent deployment, and decentralised development (Dragoni et al., 2017; Newman, 2019), these same characteristics complicate traditional measurement approaches (Bogner, Fritzsch, Wagner and Zimmermann, 2019; Abgaz et al., 2023).

One major issue lies in the lack of system-wide visibility across services. Traditional metrics often assume centralised codebases and shared state, whereas microservices are distributed and independently managed (Bogner, Fritzsch, Wagner and Zimmermann, 2019; Dragoni et al., 2017). This makes it difficult to apply system-level quality metrics, such as coupling and cohesion, since services interact asynchronously and are deployed across diverse environments (Newman, 2019).

Additionally, microarchitectural assessment tools are rarely used in practice. In their empirical study, Bogner, Fritzsch, Wagner and Zimmermann (2019) found that industry professionals predominantly rely on tools like SonarQube to assess local code-level metrics such as test coverage, code smells, and cyclomatic complexity while neglecting architectural-level metrics that assess service interaction, granularity, and modularity. This is especially concerning given that most reported issues related to architectural technical debt, such as inappropriate service boundaries and poor cohesion (Bogner, Fritzsch, Wagner and Zimmermann, 2019).

Another major concern is the absence of service-oriented metrics. While developers recognise the importance of design principles like low coupling and high cohesion, these are rarely measured explicitly during or after migration (Bogner, Fritzsch, Wagner and Zimmermann, 2019). In many cases, the process of "service cutting" deciding how to decompose a monolith is described as a manual and error-prone activity with little tool support (Abgaz et al., 2023). This lack of concrete measurement makes it difficult to validate whether decomposition decisions actually improve the architecture.

There are also challenges in interpreting local metrics in a distributed system. Metrics like test coverage or LOC are meaningful in a monolith but become harder to contextualise when applied to individual microservices Bogner, Fritzsch, Wagner and Zimmermann (2019) Developers often struggle to relate local service metrics to global quality goals, which hinders efforts to ensure consistency and maintainability across the entire system (Dragoni et al., 2017).

Finally, industry tooling often fails to support the evaluation of high-level architecture. Despite acknowledging the value of system-centric views, most teams lack tools to monitor cross-service dependencies, architectural violations, or service boundary degradation (Bogner, Fritzsch, Wagner and Zimmermann, 2019; Abgaz et al., 2023). This leads to an overreliance on local metrics, which may not reflect the broader quality concerns relevant to microservice architecture.

The challenges in measuring microservices stem from the mismatch between traditional software metrics and the decentralized, asynchronous, and independently evolving nature of microservices (Abgaz et al., 2023). These challenges are more prominent during the migration phase, when teams must evaluate the quality of both the legacy monolith and the emerging service-based system. As noted by Bogner, Fritzsch, Wagner and Zimmermann (2019); Abgaz et al. (2023), there is a pressing need for metrics and tools that address architectural quality at scale, support service-oriented measurement, and enable consistent evaluation during system evolution.

2.3 Code Smells and Their Relevance to Quality

Code smells are widely recognised as indicators of underlying design or structural problems in software systems (Fowler, 1999; Yamashita and Moonen, 2013). Although they may not directly cause faults or failures, their presence has been shown to correlate with decreased maintainability, reduced testability, and greater long-term development effort (Tufano et al., 2015). The term "smell" is deliberately metaphorical, it captures the intuition that, while code may execute correctly, there are design choices that warrant scrutiny (Fowler, 1999). Since their introduction, code smells have evolved from informal heuristics into formalised design anti-patterns supported by detection techniques and automated tools (Lanza and Marinescu, 2006; Moha et al., 2010). Today, they play a critical role in static analysis, refactoring recommendation systems, and architectural assessment (Fontana et al., 2016; Khomh et al., 2009). This section reviews foundational work on object-oriented smells, as first described by (Fowler, 1999), and contrasts them with the architectural smells specific to microservice systems, as empirically derived by (Taibi and Lenarduzzi, 2018).

2.3.1 Foundational Work on Code Smells

The term "code smell" was popularized by Fowler (1999) in his seminal work on refactoring. Fowler listed about 22 common smells (e.g. Long Method, Large Class/God Class, Feature Envy) that signal when refactoring may be needed. He drew on earlier ideas such as Riel (1996) discussion of design flaws, but it was Fowler who coined the smell metaphor for code issues. Fowler himself did not give formal detection rules; instead he emphasized that smells are subjective heuristics, developers must learn to recognize them by experience. Later researchers built on Fowler's catalog by formalizing detection. For example, (Lanza and Marinescu, 2006) defined disharmonies using object-oriented metrics and concrete thresholds. In general, foundational work on smells (Lanza and Marinescu, 2006) combined Fowler's intuitive definitions with quantifiable metrics: many classic smells (God Class, Long Method, etc.) were ultimately defined in terms of size, complexity, coupling, cohesion and other code metrics. These early efforts created the basis for modern smell detection tools and studies.

Table 2.1: Code Smells and their Descriptions (M. Fowler, 1999)

Smell	**Description**
Alternative classes with different interfaces	Two classes appear different on the outside, but are similar on the inside
Comments	Comments should describe why the code is there not what it does
Data class	Classes should not contain just data, they should contain methods as well
Data clumps	Data that belong together should be amalgamated rather than remain separated
Divergent change	Changes to code should be kept local; too many diverse changes indicate poor structure
Duplicated code	Eradicate duplicated code whenever possible
Feature Envy	Class features that use other "class" features should be moved those other classes
Inappropriate intimacy	Classes should not associate with other classes excessively
Incomplete library class	Avoid adding a method you need (and which does not exist in a library class) to a random class
Large class	A class has too many methods
Lazy class	A class is doing too little to justify its existence
Long method	A method is too large; it should be decomposed
Long parameter list	A method has too many parameters
Message chains	Avoid long chains of message calls
Middle man	If a class is delegating too much responsibility, should it exist?
Parallel inheritance hierarchies	When you make a subclass of one class, you need to make a subclass of another
Primitive obsession	Overuse of primitive types in a class
Refused bequest	If inherited behaviour is not being used, is inheritance necessary?
Shotgun surgery	Avoid cascading changes; limit the number of classes that need to be changed
Speculative generality	Code should not be added for "just in case" scenarios—it should solve current problems

Continued on next page

Table 2.1 – continued from previous page

Smell	Description
Switch statements	Polymorphism should be used instead of large switch statements
Temporary field	Unnecessary fields

2.3.2 Hierarchy of Code smells

The hierarchy of code smells provides a structured framework for grouping related smells and understanding their relationships. Mantyla et al. (2003) taxonomy is one of the most widely referenced, classifying smells into categories such as Bloaters, Object-Oriented Abusers, Change Preventers, Dispensables, Encapsulators, Couplers, and Others. These groupings help link smells to the underlying design problems they represent, for example, Bloaters indicate excessive size or complexity, while Couplers point to problematic dependencies between modules Mantyla et al. (2003). Such categorisation supports both practitioners in prioritising refactoring and researchers in structuring empirical studies.

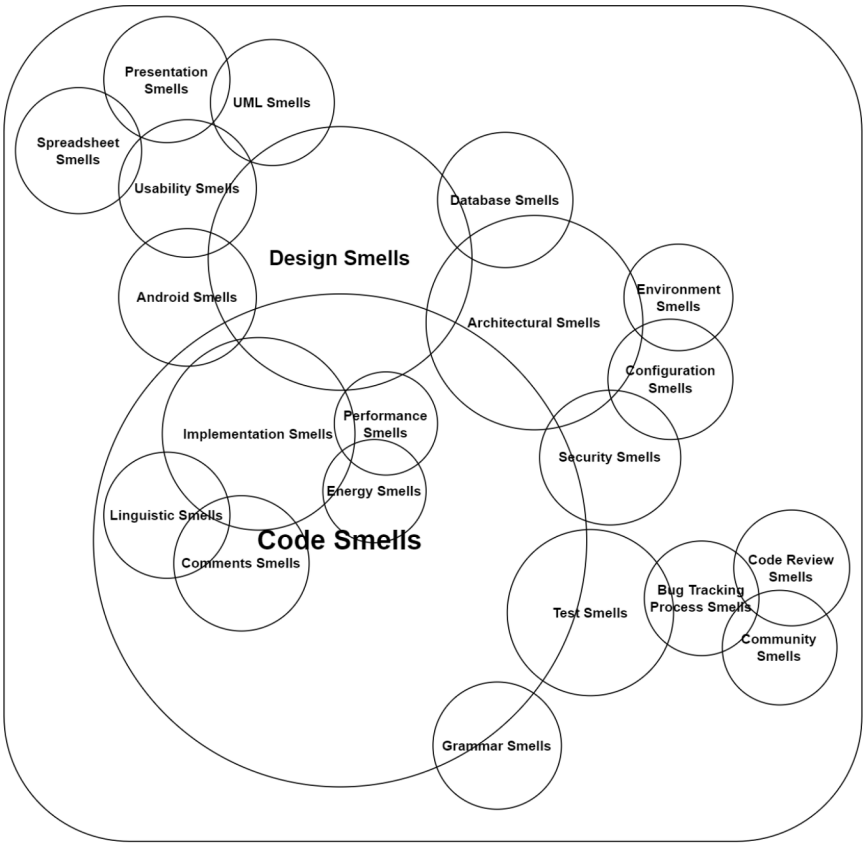


Figure 2.3: Vienna diagram of Hierarchies of Bad Smells (Jerzyk and Madeyski, 2023)

Jerzyk and Madeyski (2023) expand on this work by identifying 22 distinct smell hierarchies across the literature (Figure 2.3), revealing substantial inconsistencies in naming, definitions, and classification. They note that some smells such as Parallel Inheritance Hierarchies are placed in different categories depending on the source, and that “bad smell” is often used interchangeably with “code smell,” “design smell,” or “antipattern.” To address this, they propose treating categories as flexible, feature like labels rather than rigid bins, allowing a smell to belong to multiple relevant groups (Jerzyk and Madeyski, 2023).

2.3.3 Microservice Specific Code Smells

While Fowler’s smells focus on low-level code structures such as classes and methods, modern systems especially those built using microservice architectures introduce new categories of design degradation. (Taibi and Lenarduzzi, 2018) empirically identified a catalog of 11 microservice-specific "bad smells" based on interviews with 72 experienced practitioners. These smells emerge from architectural and operational decisions rather than internal code complexity.

Unlike foundational smells, which are mostly structural and universal, microservice smells are contextual, they arise in the presence of distributed systems, team autonomy, and service boundaries. Table 2.2 lists these smells along with their descriptions.

Table 2.2: Code Smells related to Microservices (Taibi and Lenarduzzi, 2018)

Smell	**Description**
API Versioning	APIs are not semantically versioned, leading to potential incompatibility and client-service integration issues.
Cyclic Dependency	Microservices call each other in a cycle, harming modularity and reuse.
ESB Usage	Microservices use a centralized Enterprise Service Bus (ESB), reducing independence and increasing complexity.
Hard-Coded Endpoints	Services rely on fixed IPs or ports, breaking location transparency and complicating deployment.
Inappropriate Service Intimacy	A service accesses private data of another, increasing coupling and breaking encapsulation.
Continued on next page	

Table 2.2 – continued from previous page

Smell	Description
Microservice Greedy	Too many microservices are created for minor features, resulting in over-fragmentation and poor maintainability.
No API Gateway	Direct communication between services or consumers without an API Gateway leads to unmanageable integration.
Shared Libraries	Common libraries used across services cause tight coupling and coordination overhead.
Shared Persistency	Multiple services access the same database or tables, reducing independence and increasing change impact.
Too Many Standards	Excessive variety of tech stacks causes onboarding, maintenance, and skill-sharing problems.
Wrong Cuts	Services are split by technical layers instead of business capabilities, leading to high coupling and complexity.

2.4 Service Cutter as a Blueprint for Reverse Engineering Decomposition Models

Transitioning a monolithic application to a microservice architecture is a complex reverse engineering task, with a primary challenge being the identification of logical service boundaries (Abgaz et al., 2023). To guide this process, researchers have developed various techniques that often rely on an intermediate representation, an abstract model of the system's structure and behavior to enable systematic analysis (Abgaz et al., 2023).

A foundational, tool-supported method in this area is the Service Cutter framework (Gysel et al., 2016). It was an attempt to move beyond purely qualitative decomposition strategies by introducing a structured, criteria-based process. The framework analyzes a system based on a predefined catalog of 16 coupling criteria to propose potential "service cuts" that maximize internal cohesion while minimizing inter-service coupling (Gysel et al., 2016; Dragoni et al., 2017).

Crucially for this dissertation, Service Cutter's primary contribution is its data model, which serves as a blueprint for the reverse engineering process. To perform its analysis, the tool requires a detailed, structured input model of the monolith, typically provided as JSON files (Gysel et al., 2016; Cojocaru et al., 2019). At its most granular level, this model is composed of "nanoentities" the indivisi-

ble units of data, operations, and artifacts within the system (Gysel et al., 2016). The process of identifying these nanoentities and mapping their relationships from an existing codebase is precisely the reverse engineering task undertaken in this study's methodology (Chapter 3.7). Therefore, Service Cutter's data model provides an established and effective template for the types of artifacts that must be systematically extracted to support automated architectural analysis (Gysel et al., 2016).

However, generating a decomposition is only the first step; validating its architectural quality is equally critical. This is the role filled by the metric-based tools that are the focus of this research. Once a system's intermediate representation has been reverse-engineered, following the principles established by Service Cutter's data model, it can be used as direct input for a static analysis framework like MicroValid (Cojocaru et al., 2019). Simultaneously, the running system can be analyzed using a dynamic tool like the one proposed by Bogner, Schlinger, Wagner and Zimmermann (2019). This establishes the core workflow of this dissertation: using an established decomposition model as a guide for reverse engineering, and then leveraging the resulting artifacts to conduct a comparative study of static and dynamic validation tools.

2.5 Static and Runtime Metric Tools for Assessing Microservices

Assessing the quality of microservice architectures requires metrics that are both relevant to service-level modularity and applicable to real-world systems. Among the numerous available metric frameworks, MicroValid (Cojocaru et al., 2019) and the runtime metric calculator by Bogner, Schlinger, Wagner and Zimmermann (2019) were selected for this study because they directly measure coupling and cohesion at the service level. MicroValid provides a static analysis perspective by computing structural metrics from source code, while Bogner et al., approach offers a dynamic perspective based on runtime execution traces. This combination enables a complementary, side-by-side evaluation of static and runtime characteristics.

2.5.1 MicroValid Framework

MicroValid assesses the quality of microservice decompositions using a suite of static analysis metrics targeting three fundamental software architecture principles: granularity, coupling, and cohesion (Cojocaru et al., 2019). Each dimension

is evaluated through dedicated tests, using the coefficient of variation (Cv) to identify inconsistencies across services and normalize results on a 0–10 scale (Figure 2.4) (Cojocaru et al., 2019).

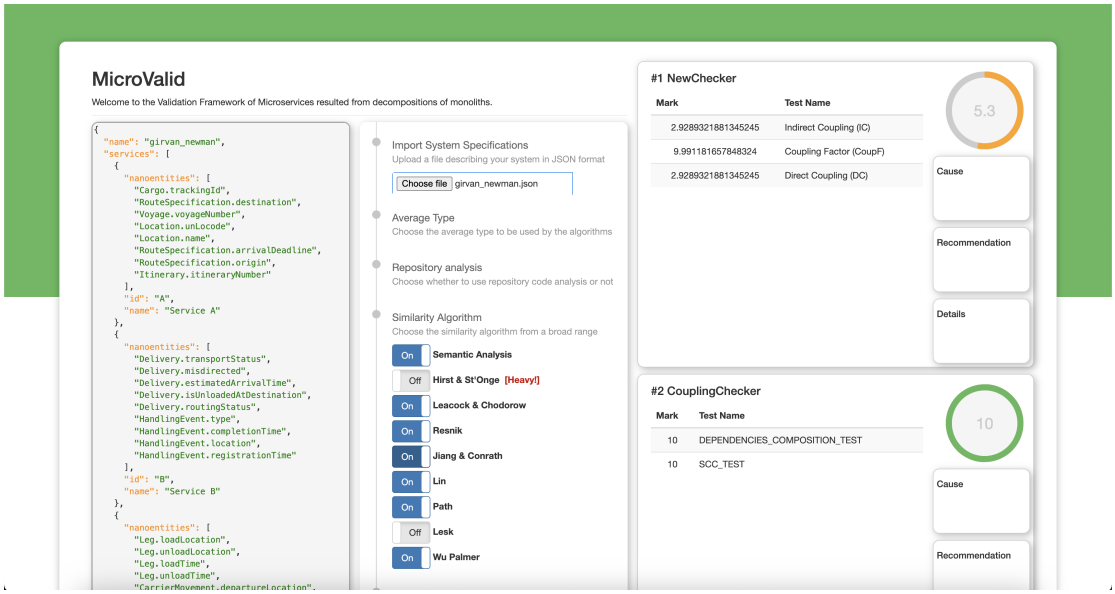


Figure 2.4: MicroValid Sample Result

The coefficient of variation is a standardized measure of dispersion, calculated as the ratio of the standard deviation (σ) to the mean (μ):

$$C_v = \frac{\sigma}{\mu}$$

A low C_v indicates that the services are uniform with respect to the measured attribute (e.g., all have a similar size), which is generally desirable. A high C_v signals an imbalance or outlier. The framework maps the calculated C_v to a user-friendly 0–10 scale, where a lower C_v corresponds to a higher score. The final scores are classified into three (Cojocaru et al., 2019):

- **Good:** $(7.5, 10.0]$
- **Acceptable:** $(5.0, 7.5]$
- **Failed:** $[0.0, 5.0]$

Granularity

Granularity refers to the size and scope of individual microservices, ideally ensuring that services are neither too coarse-grained (monolith-like) nor excessively fine-grained (leading to communication overhead) (Kalske et al., 2017). MicroValid captures this attribute through two primary tests: Nanoentities Composition Test and the Lines of Code (LoC) Test.

Nanoentities Composition Test

This test evaluates the balance of service size based on the number of "nanoentities" (e.g., class fields or attributes) within each service (Cojocaru et al., 2019).

- **Calculation:** It computes the C_v of the list of nanoentity counts for all microservices.
- **Interpretation:** A high score (approaching 10) indicates that services are of comparable size, suggesting balanced granularity. A low score reveals significant size disparities.

Lines of Code (LoC) Test

This provides a code-level size assessment.

- **Calculation:** It follows the same logic as the nanoentities test but uses the lines of code per microservice as its input data.
- **Interpretation:** Similar to the above, a high score is better, signifying that the implementation effort and complexity are evenly distributed.

Coupling

Coupling is a critical quality dimension that reflects the degree of independence between microservices. High coupling can compromise service autonomy, making changes or deployments riskier (Kalske et al., 2017). MicroValid assesses coupling using two tests: the Dependencies Composition Test and the Strongly Connected Components (SCC) Test.

Dependencies Composition Test

This test measures how evenly distributed the outward dependencies are among services (Cojocaru et al., 2019).

- **Calculation:** It constructs a dependency graph and calculates the C_v for the list of outward dependency counts for each microservice.
- **Interpretation:** A high score is desirable, as it implies that no single service is acting as a central, overly-coupled hub.

Strongly Connected Components (SCC) Test

This test detects cyclical dependencies, where a group of services are tightly interdependent (Cojocaru et al., 2019).

- **Calculation:** It uses Tarjan's algorithm on the dependency graph to identify all SCCs. The score is calculated as the ratio of identified SCCs to the total number of services, mapped to the 0-10 scale.
- **Interpretation:** The ideal score is 10, which occurs when the score ratio is 1 (i.e., every service is its own SCC, and there are no cycles). A lower score indicates the presence of problematic cyclic dependencies that should be resolved, often by merging the services within the cycle.

Cohesion

Cohesion describes how closely the elements within a service are related to one another. High cohesion ensures that a microservice encapsulates a single, focused responsibility, which is essential for maintainability and scalability (Kalske et al., 2017). MicroValid assess cohesion using the following tests;

Entities and Responsibilities Composition Tests

These tests assess the distribution of entities (e.g., classes) and use case responsibilities across services (Cojocaru et al., 2019).

- **Calculation:** They both use the coefficient of variation (C_v) on the counts of entities or responsibilities per service.
- **Interpretation:** High scores are better, indicating a balanced distribution of responsibilities and business concepts.

Relation Composition Test

This test analyzes the "published language" (shared data) in the communication paths between services to identify potential bottlenecks (Cojocaru et al., 2019).

- **Calculation:** It computes the C_v on the number of shared entities per relation. The test automatically fails (scores 0) if it detects the same entity being duplicated across multiple relations, as this indicates low cohesion.
- **Interpretation:** A high score suggests that the data exchange between services is balanced.

Semantic Similarity Test

This test assesses cohesion by measuring the lexical relatedness of the names of components within a service (Cojocaru et al., 2019).

- **Calculation:** It employs up to eight different WordNet-based similarity algorithms (e.g., Wu & Palmer, Lesk). The final score is the average of the selected algorithms' outputs, mapped to the 0-10 scale.
- **Interpretation:** A high score (approaching 10) signifies strong semantic coherence (e.g., a service named "PaymentProcessor" contains entities like "CreditCard" and "Transaction"), implying high functional cohesion.

MicroValid (Cojocaru et al., 2019) was selected as the static analysis tool for this study because it directly computes service-level coupling and cohesion from source code. Unlike general-purpose static metric suites such as Chidamber and Kemerer (1994) and Sarkar et al. (2005), which target class or package-level designs, MicroValid's metrics are defined at the microservice boundary, making them directly relevant for this research. Its open-source implementation and well-documented configuration also allow full reproducibility and automation across multiple systems, which was a key requirement for the comparative analysis in this dissertation.

2.5.2 Bogner's Runtime Calculator

Bogner, Schlinger, Wagner and Zimmermann (2019) propose a modular, dynamic analysis tool for evaluating the maintainability of microservice-based systems by extracting runtime metrics through distributed tracing. This approach directly addresses key limitations of static analysis methods in modern service-oriented architectures, where decentralization and heterogeneity often obscure architectural characteristics at the code level (Bogner, Schlinger, Wagner and Zimmermann, 2019).

The tool is designed around three core principles, simplicity, extensibility, and broad applicability and is implemented as a Java-based command-line interface (CLI). It follows a pipes-and-filters architectural style (Figure 2.5), consisting of three modular components: Integrators (for data ingestion), Metrics (for analysis), and Exporters (for reporting). This structure allows developers to flexibly add or adapt data sources and metrics without modifying the core logic (Bogner, Schlinger, Wagner and Zimmermann, 2019).

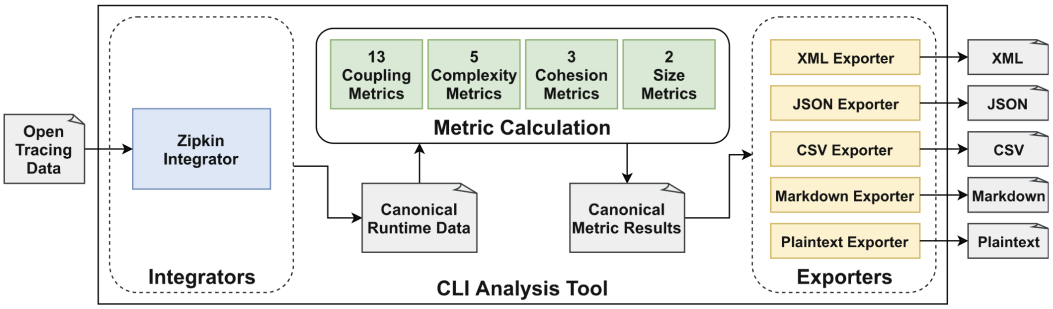


Figure 2.5: Runtime metric tool architecture

In its prototype, the tool integrates with Zipkin, a widely used distributed tracing system that adheres to the OpenTracing standard. Zipkin tracers embedded in each service record invocation data, which is then transformed into an internal canonical model representing services, operations, and their dependencies. This model is used to compute a wide range of architectural metrics using efficient graph-based operations (Bogner, Schlinger, Wagner and Zimmermann, 2019).

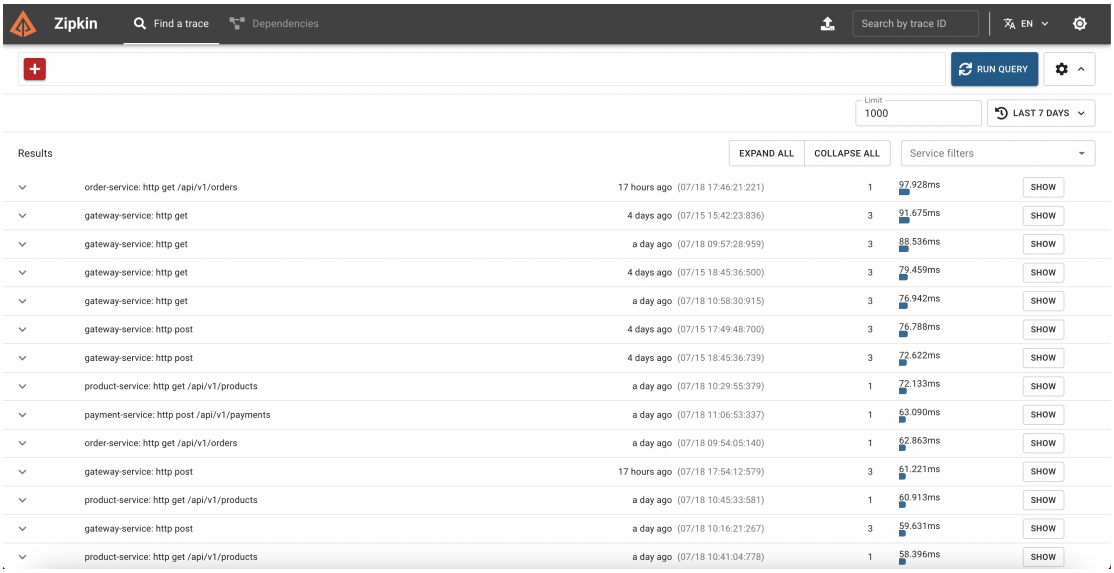


Figure 2.6: Sample Zipkin Dashboard

The authors implemented 23 metrics, spanning architectural concerns such as coupling (e.g. Service Coupling Factor, Mean Absolute Coupling in the System), cohesion (e.g. Service Interface Data/Usage Cohesion), complexity (e.g. Total Response for Service), and size (e.g. Number of Services, Weighted Service Interface Count). These metrics were selected and adapted from a systematic review of service-based maintainability metrics (Bogner et al., 2017).

Coupling Metrics

These metrics quantify the interconnections between services. In general, lower values are better (Bogner et al., 2017).

Services Interdependence in the System (SIY)

This metric measures cyclic dependencies at the system level (Bogner et al., 2017).

- **Calculation:** A raw count of all pairs of services $\langle S_1, S_2 \rangle$ that are bi-directionally dependent (i.e., S_1 calls S_2 and S_2 calls S_1).
- **Interpretation:** The range is $[0, \infty)$, with an ideal value of 0. Any value greater than 0 indicates a direct cyclic dependency that should be investigated.

Absolute Importance of the Service (AIS)

This identifies how critical a service is to others (its "fan-in") (Bogner et al., 2017).

- **Calculation:** A count of the number of unique consumer services that invoke a given service S.
- **Interpretation:** The range is $[0, N-1]$, where N is the total number of services. There is no strict upper bound, but a very high AIS relative to other services identifies a potential bottleneck or a "god" service that is central to the system.

Absolute Dependence of the Service (ADS)

This measures how reliant a service is on others (its "fan-out") (Bogner et al., 2017).

- **Calculation:** A count of the number of unique services that a given service S invokes.
- **Interpretation:** The range is $[0, N-1]$. A high ADS suggests a service may have too many responsibilities or is overly dependent on others, making it fragile.

Absolute Criticality of the Service (ACS)

This is a composite metric to pinpoint the most problematic services (Bogner et al., 2017).

- **Calculation:** The product of a service's fan-in and fan-out: $ACS(S) = AIS(S) \times ADS(S)$.
- **Interpretation:** The range is $[0, \infty)$. A high ACS score highlights services that are both highly depended upon and highly dependent, making them the riskiest points of failure and maintenance.

Cohesion Metrics

These metrics assess the functional relatedness of operations within a single service. In general, higher values are better (Bogner et al., 2017)

Service Interface Data Cohesion (SIDC)

This measures cohesion based on the data types used by a service's operations (Bogner et al., 2017).

- **Calculation:** The ratio of operations sharing common parameter data types to the total number of distinct data types in the service's interface.
- **Interpretation:** The range is $[0, 1]$. A value of 1 is ideal, indicating that all operations manipulate a common set of data structures, suggesting a highly cohesive service.

Service Interface Usage Cohesion (SIUC)

This measures cohesion based on how clients use the service's operations (Bogner et al., 2017).

- **Calculation:** The ratio of the actual number of operation invocations by all clients to the maximum possible number of invocations (number of clients $\times$ number of operations).
- **Interpretation:** The range is $[0, 1]$. A score of 1 is ideal, meaning every client uses every operation, suggesting the service offers a single, indivisible set of functionalities. A low score suggests the service could be split.

Total Service Interface Cohesion (TSIC)

This provides an aggregated cohesion score (Bogner et al., 2017).

- **Calculation:** The normalized sum (average) of SIDC and SIUC: $TSIC(S) = (SIDC(S) + SIUC(S))/2$.
- **Interpretation:** The range is $[0, 1]$. It provides a single, holistic measure of interface cohesion, where a value closer to 1 is better.

Size Metric

Weighted Service Interface Count (WSIC)

A more meaningful measure of service size than LoC in a distributed environment (Bogner et al., 2017).

- **Calculation:** A count of the number of operations exposed by a service's interface. The "weighted" aspect allows for assigning higher values to more complex operations (e.g., those with many parameters), though the default weight is 1.
- **Interpretation:** The range is $[0, \infty)$. There is no ideal absolute value. This metric is most useful for relative comparison. By calculating the system-wide average ($WSIC_{AVG}$), one can identify outlier services that are significantly larger or smaller than their peers.

Bogner, Schlinger, Wagner and Zimmermann (2019) runtime metric framework was chosen as the dynamic analysis counterpart to MicroValid (Cojocaru et al., 2019) because it applies the same quality concepts, coupling and cohesion at the service level, but derives them from runtime execution traces rather than source code. This alignment of measurement targets enables a direct comparison between static and dynamic perspectives, something not possible with other runtime-oriented tools such as MSANose, which focuses on detecting architectural smells without producing comparable quantitative scores. Bogner et al., approach is open-source, modular, and specifically designed for microservice architectures, making it well-suited for empirical research alongside MicroValid.

2.6 Alternative and Foundational Metric Frameworks

2.6.1 Chidamber and Kemerer (C&K) Metrics Suite

The Chidamber and Kemerer (C&K) metrics suite is a foundational framework for evaluating object-oriented software design. It provides a set of quantitative indicators that assess key architectural qualities such as complexity, coupling, and cohesion, which are critical for maintainability, testability, and extensibility (Chidamber and Kemerer, 1994). These metrics are widely used in both academic research and industrial practice to guide refactoring decisions and improve software quality.

Table 2.3: Software Metrics and their Relevancy to Microservices

Metric	**Description**	**Relevance to Microservices**	**Reason for Relevance**
WMC (Weighted Methods per Class)	Measures class complexity based on number/ complexity of methods.	Maybe	WMC can guide splitting large services into smaller, manageable ones.
DIT (Depth of Inheritance Tree)	Indicates inheritance levels from a base class.	No	
NOC (Number of Children)	Number of immediate subclasses of a class.	No	
CBO (Coupling Between Object Classes)	Number of classes to which a class is coupled.	Yes	Loose coupling is a key principle in microservices; CBO can help assess inter-service dependencies.
RFC (Response For a Class)	Number of methods that can be invoked in response to a message.	Maybe	High RFC may indicate a "God class", microservices should aim for focused, low-responsibility services.
LCOM (Lack of Cohesion of Methods)	Measures cohesion by assessing how related the class's methods are.	Yes	High cohesion is desirable in microservices; LCOM helps ensure services are functionally cohesive.

While the C&K metrics are foundational in object-oriented software analysis, they are not designed for nor do they translate well in evaluating microservice architectures. This is primarily due to a mismatch in abstraction levels: C&K metrics operate at the class level within object-oriented programming, whereas microservices are composed at the service level, often aggregating multiple classes or modules.

2.6.2 Sarkar Architectural Metrics

Sarkar et al. (2005) developed a set of quantitative metrics aimed at evaluating the modularization quality of large-scale object-oriented software systems, with particular emphasis on the role of APIs. These metrics are designed to measure multiple aspects: the degree to which intermodule interactions are mediated and constrained through well-defined APIs, the cohesiveness and clarity of the APIs themselves, and the extent of dependencies created by inheritance, associations, and other object-oriented relationships. By doing so, the metrics provide a comprehensive assessment of how well the modules are decoupled, maintainable, and extendable, especially considering the complex dependencies that can develop over time in large, evolving software systems.

Table 2.4: Module-Level Metrics and their Relevance to Microservices

Metric	**Description**	**Relevance to Microservices**	**Reason for Relevance**
Module Interaction Index	Measures how well other modules utilize a module's API functions (Sarkar et al., 2005).	Yes	Indicates API-driven use; microservices rely heavily on well-defined API usage patterns and service consumption.
Non-API Function Closedness Index	Measures how much a module prevents non-API methods from being invoked externally (Sarkar et al. 2008).	Maybe	Good for encapsulation; microservices also need internal methods hidden, but inter-service calls often use APIs anyway.
API Function Usage Index	Measures percentage of a module's API functions used by others (Sarkar et al., 2005).	Yes	Highlights lean service design; microservices favour small, well-used APIs over bloated interfaces.

Continued on next page

Table 2.4: Module-Level Metrics and their Relevance to Microservices (continued)

Metric	**Description**	**Relevance to Microservices**	**Reason for Relevance**
Module Size Uniformity Index	Evaluates consistency and variance in module sizes (Sarkar et al., 2005).	Maybe	Size uniformity aids consistency, but microservices may strategically vary in size based on domain logic.
Module Size Boundedness Index	Measures deviation of module sizes from desired norms (Sarkar et al., 2005).	Maybe	Bounded size helps maintainability, but flexibility is acceptable depending on domain responsibilities.
Cyclic Dependency Index	Measures degree of cycles among system components (Sarkar et al., 2005).	Maybe	Cycles across microservices create deployment and coupling issues.
Module Interaction Stability Index (MISI)	Assesses stability of lower-layer modules relative to higher layers (Sarkar et al., 2005).	Maybe	Useful for stable foundational services, but many microservices are ephemeral or independent.
API Cohesion (APIUC)	Measures cohesion among services within an API (Sarkar et al., 2005).	Yes	High cohesion ensures microservices support focused responsibilities and cohesive interfaces.
API Segregation (APIU)	Measures overlap or separation among client-specific APIs (Sarkar et al., 2005).	Yes	Ensures bounded contexts; microservices should avoid sharing generic APIs across unrelated clients.

Continued on next page

Table 2.4: Module-Level Metrics and their Relevance to Microservices (continued)

Metric	**Description**	**Relevance to Microservices**	**Reason for Relevance**
Class-Relationship Usage Metric (CReUM)	Assesses how often classes used together are grouped correctly across modules (Sarkar et al., 2005).	Maybe	Relevant for ensuring domain alignment and preventing distributed class logic across microservices.

Sarkar's metrics target layered or component-based architectures and are optimised for detecting violations in architectural layering. While valuable for certain architectural styles, they lack explicit support for service-level cohesion and coupling metrics that can be directly compared with runtime measurements, which was essential for this study.

2.6.3 Ntentos et al. (2020) Metrics Suite

Ntentos et al. (2020) developed a suite of metrics specifically designed to automatically assess how well a microservice architecture conforms to established coupling-related patterns and practices. Their approach moves beyond generic code-level metrics by targeting three critical sources of architectural coupling in microservices: shared databases, synchronous invocations, and dependencies on shared services. Each metric is calculated as a continuous value from 0 to 1, representing the degree to which a desirable pattern (e.g., 'Database per Service') is supported or an anti-pattern (e.g., 'Shared Database') is present. This framework provides a model-based, technology-independent way to quantify architectural health, allowing teams to systematically measure and track adherence to best practices for loose coupling.

Table 2.5: Architectural Metrics and their Relevance to Microservices

Metric	**Description**	**Relevance to Microservices**	**Reason for Relevance)**
Database Type Utilization (DTU)	Measures the ratio of dedicated databases to total DB-service links, reflecting data ownership.	Yes	Promotes service autonomy by ensuring each service owns its data, reducing hidden coupling.
Shared Database Interactions (SDBI)	Quantifies inter-service communication via shared databases.	Yes	High SDBI undermines independence and introduces tight coupling, risking cascading failures and coordination issues.
Service Interaction via Intermediary Component (SIC)	Measures how much service communication is mediated by intermediaries like brokers or gateways.	Yes	Encourages decoupling and resilience via asynchronous or intermediary-based communication.
Asynchronous Communication Utilization (ACU)	Assesses adoption of asynchronous messaging patterns like events, polling, queues.	Yes	Asynchronous patterns improve scalability, fault tolerance, and loose coupling.
Direct Service Sharing (DSS)	Captures how often services share common utility components directly.	Maybe	Sharing utilities may centralize functionality but risks tight coupling and single points of failure.
Transitively Shared Services (TSS)	Measures hidden, indirect service dependencies through shared components.	Yes	Reveals non-obvious coupling that complicates testing and deployment sequencing.

Continued on next page

Table 2.5: Architectural Metrics and their Relevance to Microservices (continued)

Metric	Description	Relevance to Microservices	Reason for Relevance
Cyclic Dependency Detection (CDD)	Binary metric indicating presence of cycles in service dependency graph.	Yes	Cycles break autonomy and deployment independence, violating key microservices principles.

The metric suite proposed by Ntentos et al. covers a broader range of qualities, including scalability and resilience, but does not provide a one-to-one mapping of service-level cohesion and coupling metrics suitable for static–dynamic comparison. Additionally, its implementation is less mature and less automated compared to MicroValid and Bogner et al., framework.

2.6.4 Al-Debagy and Martinek (2020) Metric Suite

To evaluate microservice designs, Al-Debagy and Martinek (2020) introduced a novel metrics framework that quantifies architectural quality by analyzing a system's Application Programming Interface (API). The framework targets three key attributes: granularity, cohesion, and complexity.

Table 2.6: Granularity and Cohesion Metrics for Microservices

Metric	Description	Relevance to Microservices	Reason for Relevance
Service Granularity Metric (SGM)	Evaluates granularity of microservices based on data (DGS) and functional (FGS) aspects.	Yes	Proper granularity ensures each microservice is neither too coarse (monolithic) nor too fine-grained (chatty/inefficient).

Continued on next page

Table 2.6: Granularity and Cohesion Metrics for Microservices (continued)

Metric	**Description**	**Relevance to Microservices**	**Reason for Relevance**
Data Granularity of a Service (DGS)	Measures input/output data sizes per operation to assess whether a service handles large (coarse) or small (fine) data.	Yes	Helps detect services that may violate microservices' principle of light-weight communication and independence.
Functional Granularity of a Service (FGS)	Measures how many distinct functions a microservice performs, aiming for a balanced service scope.	Yes	Overloaded services indicate weak boundaries; underloaded ones can cause coordination overhead.
Lack of Cohesion Metric (LCOM)	Assesses cohesion of methods within a service; adapted from OO class cohesion.	Yes	High cohesion ensures the microservice is focused, promoting maintainability and strong service boundaries.
Number of Operations (NOO)	Counts the total operations (methods) in a microservice, used as a proxy for functional complexity.	Yes	More operations suggest higher complexity and risk of low maintainability; design guidelines recommend fewer than 10.

2.6.5 Justification for Static and Dynamic Tool Choice

The decision to use MicroValid and Bogner et al., runtime calculator was made carefully to explore the gap between structural quality and runtime behavior of microservices. While classic metric suites like those by Chidamber and Kemerer (1994) or Sarkar et al. (2005) are well-known, they were originally developed for object-oriented systems and do not support runtime analysis. Although newer metric sets for microservices have been proposed, many of them are either theoretical or not fully implemented as working tools. For this study, it was important

to choose tools that are not only clearly documented but also ready to use in practice.

Tools such as MSA Nose while relevant to microservice analysis, is primarily focused on detecting architectural smells rather than providing a comprehensive set of quantitative metrics for coupling, cohesion, and related quality attributes. Its scope is narrower and less aligned with the goal of producing directly comparable numerical scores across both static and dynamic approaches.

SonarQube, on the other hand, is a widely used static analysis platform, but it is designed mainly for general code quality checks such as style violations, duplicated code, or cyclomatic complexity rather than for microservice-specific architectural properties. It does not natively provide service-level metrics or support architectural coupling and cohesion analysis at the granularity needed for this research.

MicroValid was chosen to represent the static analysis approach. It is designed specifically to evaluate microservice architectures using detailed metrics about coupling, cohesion, and granularity, based on the system's design or source code. On the other hand, Bogner et al., runtime analysis tool was selected to represent the dynamic analysis approach. It uses real runtime data collected through Zipkin to measure how services actually interact during execution. This makes it possible to see how the system behaves in practice, beyond what is described in its design.

This pairing of tools was chosen not because they are the only options available, but because they reflect two very different ways of thinking about software quality: one based on how the system is built, and the other on how it runs. Together, they allow for a practical and meaningful comparison between design-time expectations and real-world behavior in microservice systems.

2.7 Summary

The literature surveyed highlights that while software metrics are foundational for assessing software quality, traditional static and local metrics often fall short in capturing the architectural complexities of microservices-based systems. Early research established robust metric frameworks for monolithic and object-oriented systems, but as microservices architectures have risen to prominence, significant measurement challenges and gaps have emerged. Coupling and cohesion consistently appear as dominant quality attributes central to the maintainability and scalability of microservices. However, industry practice reveals a predominant reliance on static and code-level metrics, with limited adoption of system-centric or runtime architectural evaluation tools.

Recent advances have introduced microservice-specific metric frameworks, such

as MicroValid for static analysis and the runtime metrics tool by Bogner et al., aiming to bridge the divide between design-time intentions and operational realities. Despite these developments, there remains a gap on empirical studies directly comparing static architectural metrics with their dynamic counterparts, particularly in real-world distributed systems. This gap is critical, as effective architectural decision-making depends on robust tools and validated measurement approaches that reliably reflect operational quality.

In summary, this section establishes the theoretical and practical importance of measuring coupling and cohesion in microservices but highlights a clear need for comparative, empirical evaluation of static versus dynamic metric tools.

Chapter 3

Methodology

This chapter outlines the comprehensive research methodology employed to address the research questions posed in Chapter 1. The primary objective of this study is to conduct a comparative analysis of static and dynamic metric tools for evaluating coupling and cohesion in microservices architectures. Specifically, this research compares MicroValid (static analysis framework) with **?** runtime metrics tool (dynamic analysis) when applied to identical microservice systems.

This methodology chapter is structured to provide complete transparency and reproducibility. Following established practices in empirical software engineering research, the chapter details the research design, tool selection rationale, experimental setup, data collection procedures, and analysis methods.

3.1 Installing MicroValid

MicroValid is a validation tool that requires Java to be installed on the system. This section outlines the steps to ensure Java is installed, clone the MicroValid repository, run the application, and access the tool via a web browser

3.1.1 Prerequisites

Before proceeding with the installation of MicroValid, ensure that Java is installed on your system. You can verify the Java installation by running the following command (listing 3.1) in your terminal or command prompt:

```
1 java --version
```

Listing 3.1: Command to check Java version

Figure 3.1 below shows the installed Java version.

```
~ via C base
➜ java -version
openjdk version "22.0.2" 2024-07-16
OpenJDK Runtime Environment (build 22.0.2+9-70)
OpenJDK 64-Bit Server VM (build 22.0.2+9-70, mixed mode, sharing)
```

Figure 3.1: Java Verification

Installing Java

If Java is not installed, download and install it from the Java Download Page[1]. Follow the instructions to complete the installation and ensure that the Java executable is added to your system's PATH.

3.1.2 Cloning and Running MicroValid

MicroValid can be cloned from its GitHub repository[2]. Use the following steps to clone the repository and run the application.

Cloning the MicroValid Repository

Open a command prompt and execute the following commands (listing 3.2):

```
1 git clone https://github.com/michelcojocaru/MasterProject.git
2 cd MasterProject
```

Listing 3.2: Commands to clone MicroValid repository

Running MicroValid

Navigate to the directory containing the ValidatorApplication.java file. Open the file using Visual Studio Code (VSCode)[3]. Ensure that the Java Extension Pack is installed in VSCode to provide the necessary support for running Java applications.

To run MicroValid:

1. Open ValidatorApplication.java in VSCode.
2. Click the Run button that appears above the main method or press F5.

[1] https://www.java.com/en/download/manual.jsp
[2] https://github.com/michelcojocaru/MasterProject
[3] https://code.visualstudio.com/

3.1.3 Accessing MicroValid

Once the application is running, open a web browser and navigate to http://localhost:8040/. You should see the MicroValid dashboard (Figure 3.2) indicating that the tool is running correctly.

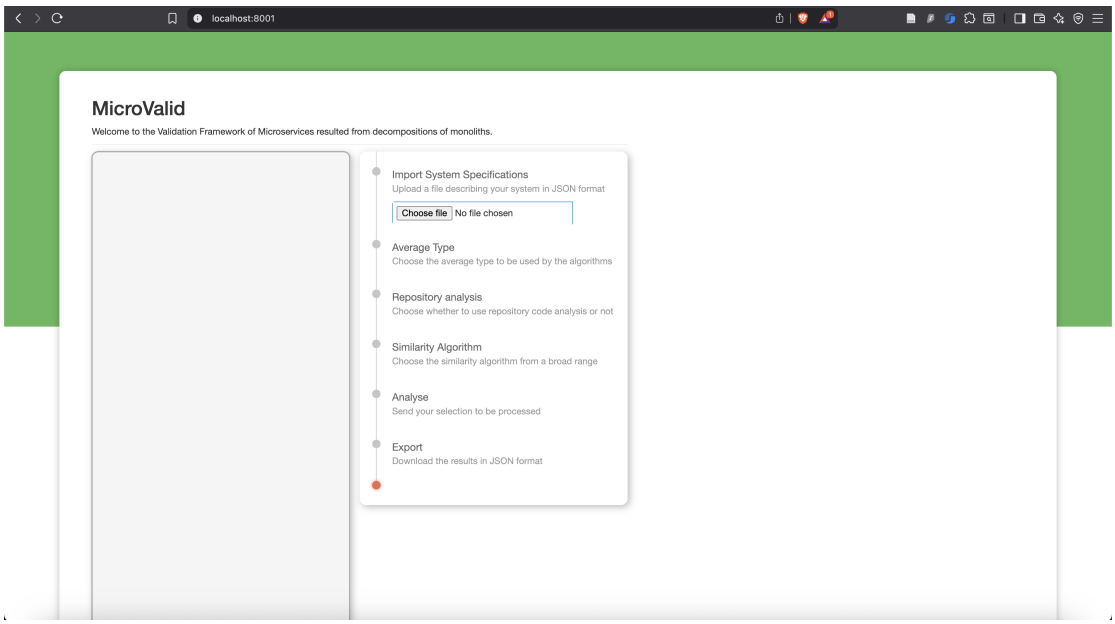


Figure 3.2: MicroValid Dashboard

By following these instructions, you will have a properly configured environment for running MicroValid.

3.1.4 MicroValid Configuration Note

Mitigating the existing bug in MicroValid

For all SEMANTIC SIMILARITY TEST calculations done in Chapter 4, the Hirst & St'Onge and Lesk algorithms were deliberately excluded for obtaining cohesion score. This decision was made because preliminary testing revealed that these specific algorithms can produce invalid scores in certain cases (Figure 3.3).

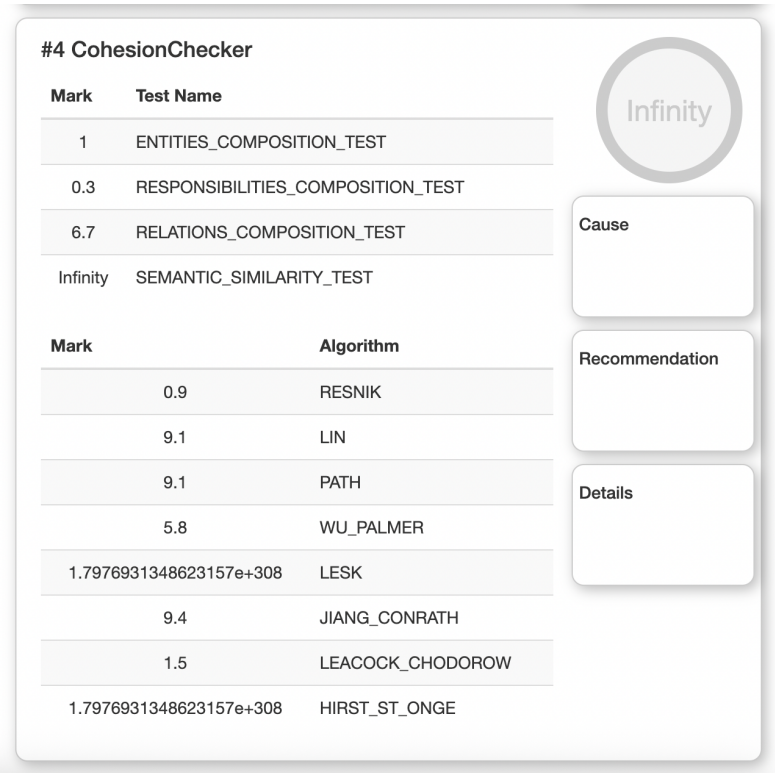


Figure 3.3: Incorrect cohesion scores when Hirst & St'Onge and Lesk algorithms turned on

3.2 Installing Microservices Runtime Analysis Tool

This tool is a Java-based CLI application designed to compute service-based maintainability metrics using operational data from microservice systems. To install and use the tool, users must first obtain the source code, then build and run it using a Java environment. This section outlines the steps required to do the same.

3.2.1 Obtaining the source code

The Microservices Runtime Analysis Tool's source code is hosted on GitHub and can be accessed at https://github.com/xJREB/microservices-runtime-analysis. To clone the repository locally, use the following command (listing 3.3) in your terminal:

```
git clone https://github.com/xJREB/microservices-runtime-analysis.git
cd microservices-runtime-analysis
```

Listing 3.3: Commands to clone the Runtime Analysis Tool repository

3.2.2 Running Microservices Runtime Analysis Tool

Start by running the following command (listing 3.4) to build the executable jar.

```
mvn clean install
```

Listing 3.4: Command to build the executable jar

The following command (listing 3.5) will only work if there is an active Zipkin instance running at http://localhost:9411.

```
    java -jar target/ms-runtime-analysis-1.0.1.jar \
    --start_date "2019/01/01 00:00:00" \
    --end_date "now" \
    --integrators "zipkin" \
    --integrator_params "zipkin:url=http://localhost:9411" \
    --exporters="xml,json,csv,human_readable,markdown" \
    --exporter_params "xml:filename=results-examples/metrics.xml" \
                      "json:filename=results-examples/metrics.json" \
                      "csv:filename=results-examples/metrics.csv" \
                      "human_readable:filename=results-examples/metrics.txt" \
                      "markdown:filename=results-examples/metrics.md"
```

Listing 3.5: Command to run the Bogner Runtime Analysis Tool

3.2.3 Installing SDKMAN!

SDKMAN! is a tool for managing parallel versions of multiple Software Development Kits (SDKs), such as Java, Maven, and Gradle, on Unix-based systems. It simplifies the process of installing, switching, and managing SDKs. Manually configuring the correct Java environment for each experiment would be time-consuming and prone to error. This ensured that each case study was compiled and executed in its intended environment, which is critical for the validity and reproducibility of the experimental results presented in Chapter 4.

To install SDKMAN![4], open your terminal and run (listing: 3.6):

```
curl -s "https://get.sdkman.io" | bash
```

Listing 3.6: SDKMAN! installation command

[4]https://sdkman.io/install

After installation, restart your terminal or run the command (listing 3.7):

```
source "$HOME/.sdkman/bin/sdkman-init.sh"
```

Listing 3.7: Command to load SDKMAN! into the shell session

You can verify the installation by typing the below command (listing 3.8):

```
sdk version
```

Listing 3.8: Check SDKMAN! version

The figure 3.4 shows the current installed version of SDKMAN!

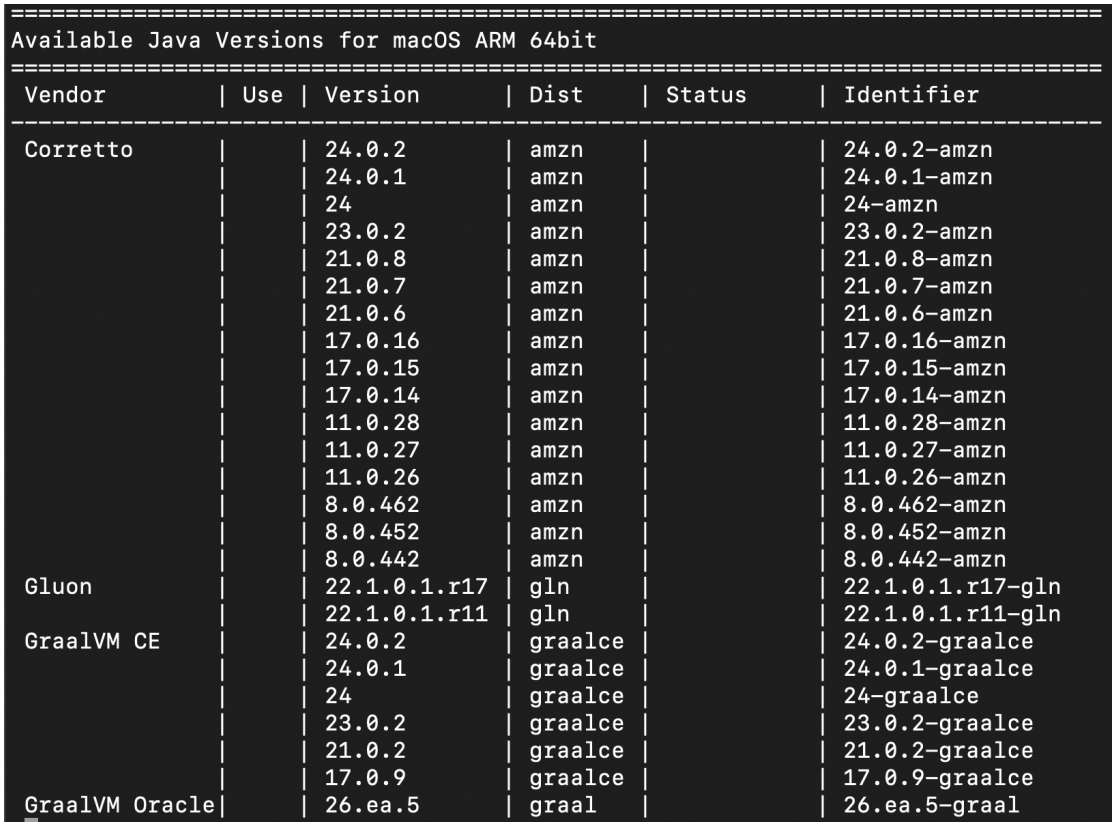
Figure 3.4: SDKMAN! Verification

You can list available Java SDKs using SDKMAN! with the following command (listing 3.9):

```
sdk list java
```

Listing 3.9: List all available Java SDK's

This will show a list of all available Java distributions and versions as described in figure 3.5



```
================================================================================
Available Java Versions for macOS ARM 64bit
================================================================================
 Vendor        | Use | Version      | Dist    | Status     | Identifier
--------------------------------------------------------------------------------
 Corretto      |     | 24.0.2       | amzn    |            | 24.0.2-amzn
               |     | 24.0.1       | amzn    |            | 24.0.1-amzn
               |     | 24           | amzn    |            | 24-amzn
               |     | 23.0.2       | amzn    |            | 23.0.2-amzn
               |     | 21.0.8       | amzn    |            | 21.0.8-amzn
               |     | 21.0.7       | amzn    |            | 21.0.7-amzn
               |     | 21.0.6       | amzn    |            | 21.0.6-amzn
               |     | 17.0.16      | amzn    |            | 17.0.16-amzn
               |     | 17.0.15      | amzn    |            | 17.0.15-amzn
               |     | 17.0.14      | amzn    |            | 17.0.14-amzn
               |     | 11.0.28      | amzn    |            | 11.0.28-amzn
               |     | 11.0.27      | amzn    |            | 11.0.27-amzn
               |     | 11.0.26      | amzn    |            | 11.0.26-amzn
               |     | 8.0.462      | amzn    |            | 8.0.462-amzn
               |     | 8.0.452      | amzn    |            | 8.0.452-amzn
               |     | 8.0.442      | amzn    |            | 8.0.442-amzn
 Gluon         |     | 22.1.0.1.r17 | gln     |            | 22.1.0.1.r17-gln
               |     | 22.1.0.1.r11 | gln     |            | 22.1.0.1.r11-gln
 GraalVM CE    |     | 24.0.2       | graalce |            | 24.0.2-graalce
               |     | 24.0.1       | graalce |            | 24.0.1-graalce
               |     | 24           | graalce |            | 24-graalce
               |     | 23.0.2       | graalce |            | 23.0.2-graalce
               |     | 21.0.2       | graalce |            | 21.0.2-graalce
               |     | 17.0.9       | graalce |            | 17.0.9-graalce
 GraalVM Oracle|     | 26.ea.5      | graal   |            | 26.ea.5-graal
```

Figure 3.5: Available Java SDK versions for macOS (via SDKMAN!)

Now you can easily install SDKs like so:

```
sdk install java 17.0.8-tem
```

Listing 3.10: Commands to clone the Runtime Analysis Tool repository

3.2.4 Docker Installation and Setup

To set up the containerization environment, Docker Desktop must be installed on the local machine.

Download Docker Desktop

Navigate to the official Docker website[5] and download the appropriate installer for your operating system (macOS, Windows, or Linux).

Install the Application

Run the downloaded installer and follow the on-screen instructions. This process will install the Docker Engine, the Docker CLI, and the Docker Desktop user interface.

[5]https://docs.docker.com/desktop/

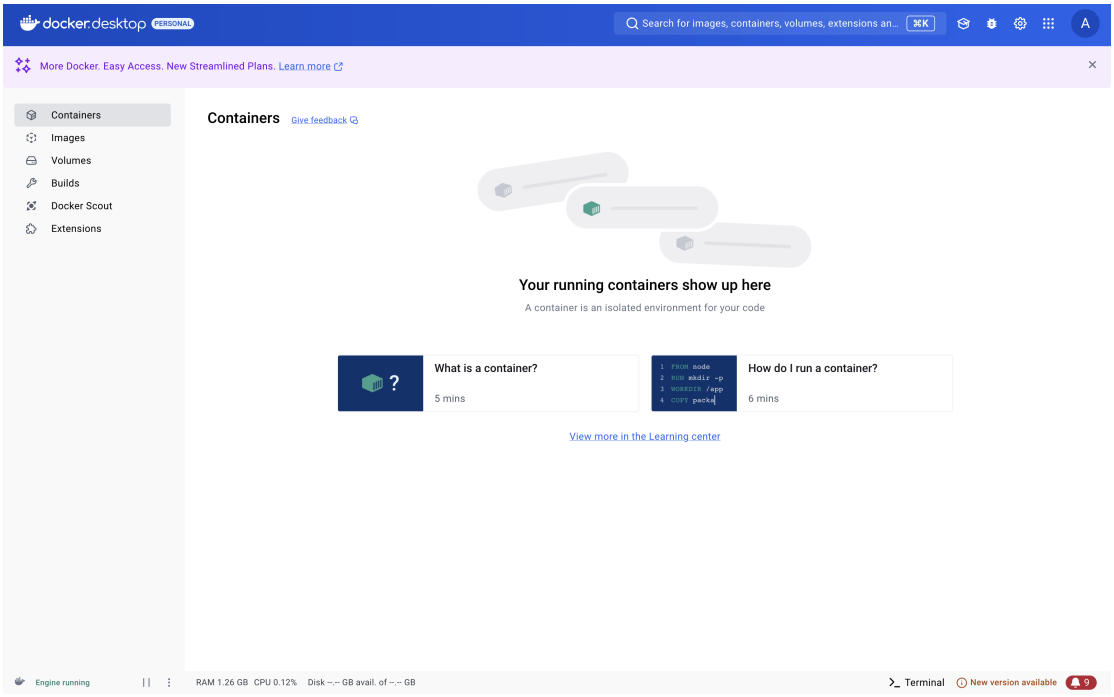


Figure 3.6: Docker Desktop containers page

Verify the Installation

Once the installation is complete and Docker Desktop is running, open a terminal or command prompt and execute the following command to verify that the Docker daemon is active and accessible (listing 3.11):

```
1 docker ps
```

Listing 3.11: Command to verify that the Docker daemon is active

This command lists all running containers. If the installation was successful, it will execute without error and display an empty table of containers.

With Docker Desktop installed and running, the environment is ready to deploy containerized applications, such as the case studies and the Zipkin server used in this research.

3.3 Configuring Zipkin with the Codebase

3.3.1 Add Zipkin and Sleuth Dependencies

These libraries enable distributed tracing and send trace data to Zipkin. Observe the below listing 3.12 - Here lines 1 - 4; shows the zipkin dependency and lines 5 - 8 shows sleuth dependency added.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

Listing 3.12: pom.xml file changes to add Zipkin and Sleuth dependencies

Configure Zipkin in application.yml

To configure Zipkin endpoint and tracing sampling rate the following changes are made to the application.yml, listing 3.13 (line 4 - 8).

The sleuth.sampler.probability: 1.0 setting ensures that 100% of requests are traced, allowing every HTTP call to be captured and sent to Zipkin.

```
spring:
  application:
    name: order-service
  zipkin:
    base-url: http://localhost:9411
    enabled: true
  sleuth:
    sampler:
      probability: 1.0
```

Listing 3.13: Configure Zipkin and Sleuth in application.yml

Running the Zipkin Instance

To install Zipkin using Docker[6], simply run the official Zipkin container from Docker Hub. This allows you to quickly set up a distributed tracing server without manual configuration. Open your terminal and execute the following command - listing 3.14:

```
docker run -d -p 9411:9411 openzipkin/zipkin
```

Listing 3.14: Commands to run the Zipkin docker instance

Now verify if the zipkin container is running properly on Docker Desktop (Figure 3.7)

[6]https://zipkin.io/pages/quickstart

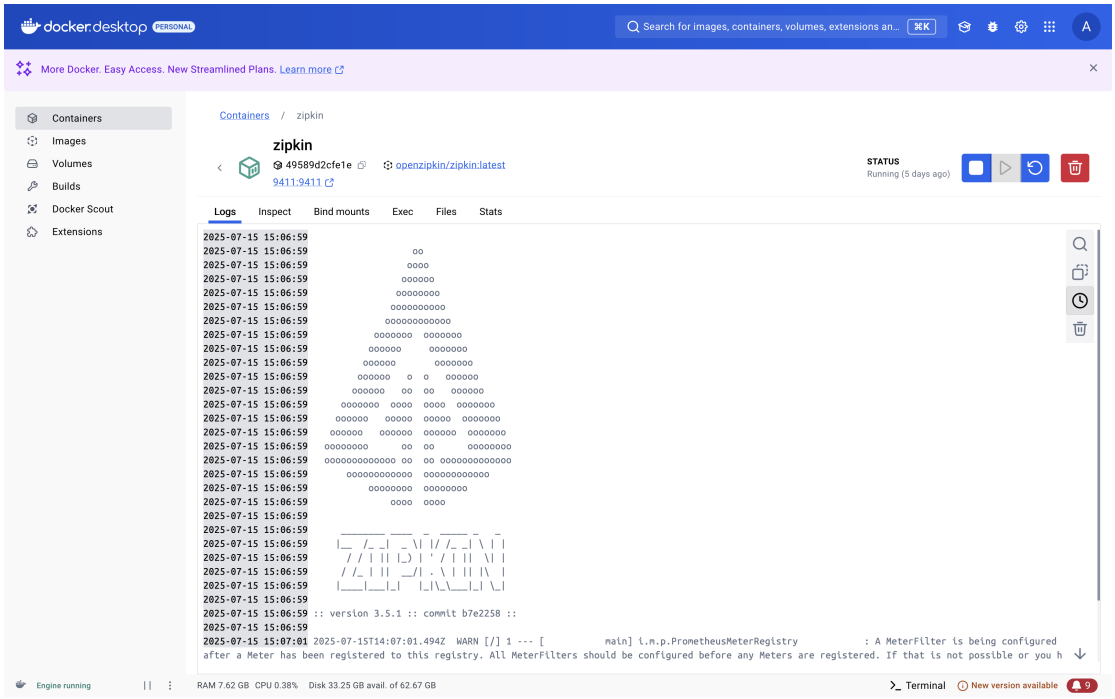


Figure 3.7: Zipkin instance running on Docker Desktop

Open the browser and go to http://localhost:9411 and we will be able to access the Zipkin dashboard (figure 3.8). Initially, there would be zero traces visible since there are no applications that send distributed tracing data to Zipkin running in the background.

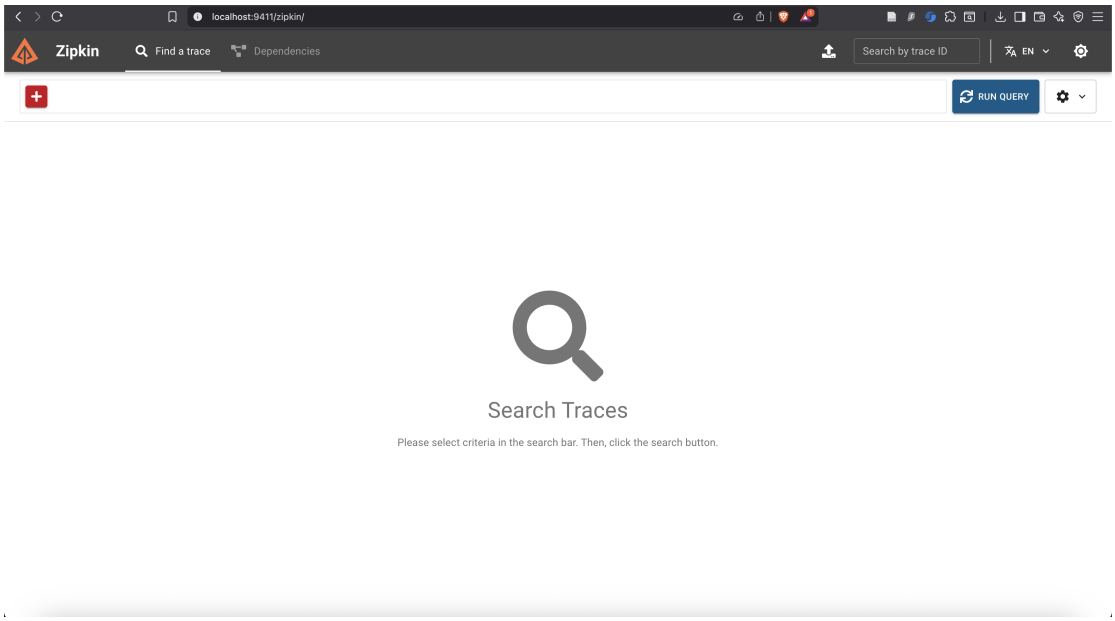


Figure 3.8: Zipkin dashboard running on http://localhost:9411

3.4 Using Graphviz for Visualization

3.4.1 Purpose of Visualization

The primary purpose of the graph-based visualization is to empirically address:
RQ1: Which software quality attributes are predominantly associated with code smells in microservice architectures?

The relationship between code smells and the software metrics that indicate them is inherently complex and many-to-many; a single smell can be indicated by multiple metrics, and one metric can point to several different smells . While a simple list could enumerate these connections, it would fail to reveal the concentration of these relationships around specific quality attributes.

Therefore, a graph was constructed to visually map this network of associations. This approach allows for a holistic observation of the entire system of relationships, making it possible to identify which types of metrics act as central hubs for the most frequently cited code smells. By revealing these clusters, the visualization provides direct, empirical evidence that the dominant quality attributes implicated by code smells are coupling and cohesion, thereby justifying their selection as the core focus of this comparative study.

3.4.2 Tool Selection and Justification

Graphviz[7] was chosen over alternatives such as Gephi, Cytoscape, and D3.js for the following reasons:

1. **Minimal setup overhead**: Graphviz is lightweight, cross-platform, and does not require a GUI or browser runtime, making it ideal for inclusion in the replication package.

2. **DOT language expressiveness**: The declarative Graph Description Language (DOT) allows precise control over node attributes, edge direction, and layout without complex scripting.

3. **High-quality static rendering**: Graphviz produces publication-ready diagrams that retain clarity even for dense networks, making it suitable for academic presentation.

[7]https://graphviz.org/documentation/

3.4.3 Graph Construction Steps

1. **Source Data Preparation:**
    - All smell–metric relationships, along with inference type (*explicit*/*inferred*) and citations, are recorded in a structured CSV file (`mapping.csv`).
    - The CSV serves as the authoritative evidence source; no relationships are added directly in the DOT file.
2. **Node Definition**
    - **Yellow circles** - Code smells.
    - **Green rectangles** - Coupling Metrics
    - **Purple rectangles** - Cohesion Metrics
3. **Edge Definition:**
    - Directed edges from smells to their indicator metrics.
    - Multiple edges may originate from a single smell to reflect multi-metric indicators.
4. **DOT File Generation:**
    - A simple script reads the CSV and outputs a Graphviz DOT file (`mapping.dot`) containing only layout and styling instructions.

```
digraph G {
  layout=neato;
  overlap=false;
  splines=true;
}
```

Listing 3.15: Graphviz layout instructions

5. **Node and Edge Styling:**
    - Visual attributes (colour, shape, font) are assigned directly in DOT syntax for clarity and consistency.

```
SF1 [label = "Duplicated Code";shape = circle;fillcolor =
    "#ffefcc";];
```

Listing 3.16: Graphviz syntax to create a node

    - Edges are declared for each smell (e.g., `Long Method - Lines of Code (LoC);`).

6. **Rendering:**

```
dot -Tsvg mapping.dot -o mapping.svg
```

Listing 3.17: Render the graph as SVG

Once all these steps are done, the resulting diagram will look like Figure 3.9.

3.5 Mapping Code Smells to Software Metrics

3.5.1 Literature-Driven Mapping

The core connections in the graph are derived from formal, metric-based detection strategies. The foundational work of Marinescu (2004) and the DECOR method detailed by Moha et al. (2009) provided the logic for rule-based detection. This was further solidified by the empirical work of Bigonha et al. (2019) and Filó et al. (2024), who used large-scale analysis to derive metric thresholds for detecting smells like Large Class, Long Method, and Refused Bequest. For highly specific smells, such as Temporary Field, the detection strategy was based entirely on the unique metrics proposed by Gupta and Singh (2020).

This analysis was extended to architectural and microservice smells, guided by the empirical study from Arcelli Fontana et al. (2019) and the SLR on architectural smells by Mumtaz et al. (2021). The metrics for these smells were sourced from the comprehensive literature review by Bogner et al. (2017); Bogner, Fritzsch, Wagner and Zimmermann (2019), the framework by Al-Debagy and Martinek (2020), and the targeted studies on microservice coupling by Zhong et al. (2023).

Each relationship in the graph was coded as either:

- **Explicit**: The source paper directly states that a specific metric is used to detect a given smell.
- **Inferred**: The connection is logically deduced from the paper's definitions of smells as violations of core design principles (e.g., high coupling) and the definitions of metrics that measure those principles.

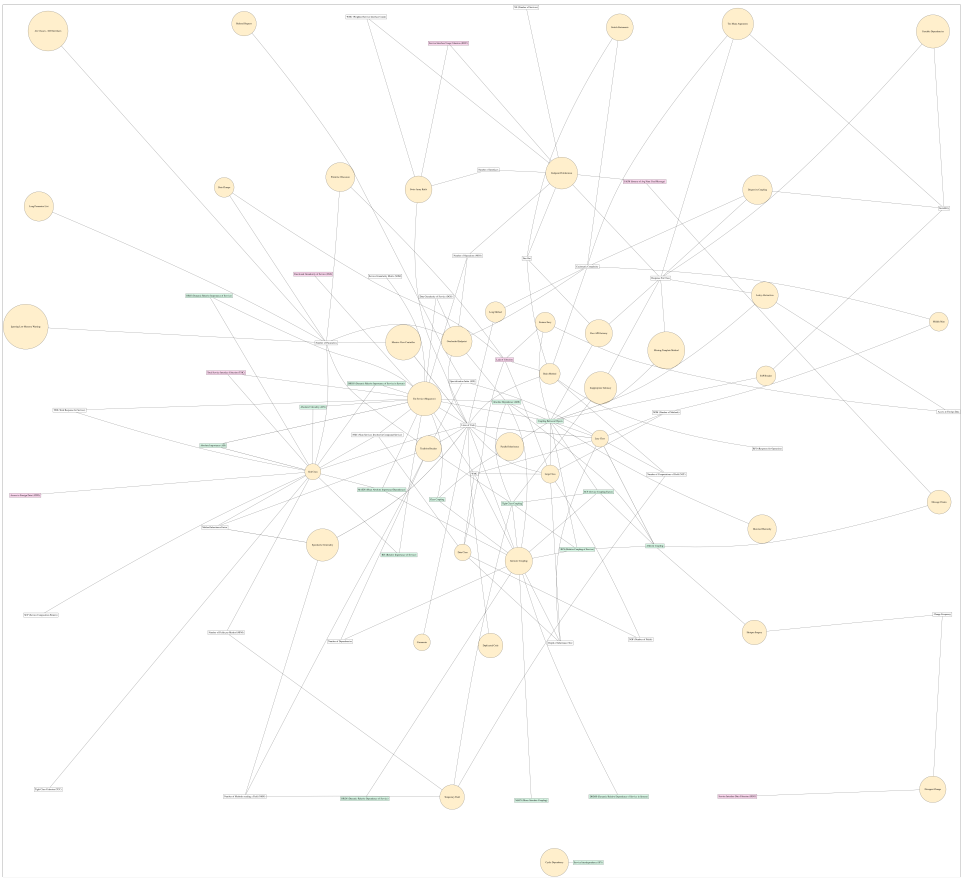


Figure 3.9: Code smell - Metric Mapping Diagram

3.5.2 Legend and Visual Encoding

This legend (Figure 3.10) explains the meaning behind the shapes and colors used in the mapping diagram to help understand the different elements and their relationships.

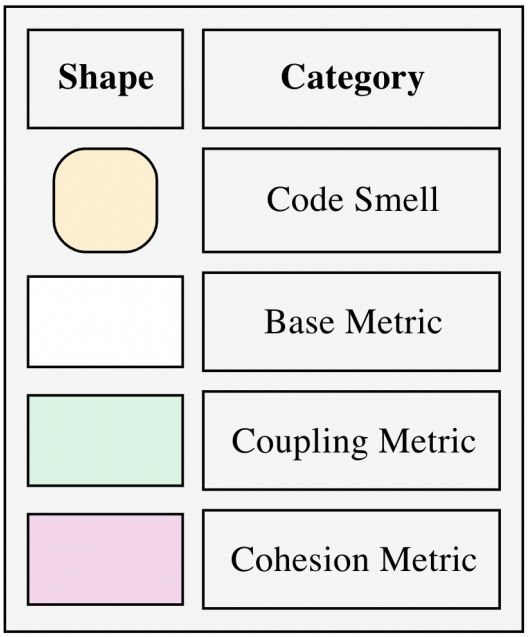


Figure 3.10: Code Smell - Metric Mapping Legend

3.5.3 Key Observations from the Mapping Diagram

A closer look at the metrics in the diagram (Figure 3.9) shows that most of them measure coupling and cohesion. In total, 56 unique metrics from the literature were analyzed

- Coupling Metrics: 18 metrics
- Cohesion Metrics: 7 metrics
- Other Quality Attributes: 31 metrics (related to size, complexity, inheritance, etc.)

When combined, coupling and cohesion metrics represent 45% (25 out of 56) of all unique metrics used to identify code smells. This large share shows that almost half of the effort in detecting code smells focuses on measuring these two core attributes, making them the most dominant quality attributes.

3.5.4 Reproducibility

The DOT source file and the corresponding SVG image are available in Github, Link: https://github.com/adarshajit/msc-thesis.

3.6 Runtime Data Collection via Zipkin

To support the dynamic analysis of coupling and cohesion metrics, distributed tracing was employed to collect real-time inter-service communication data. The tool used for this purpose was Zipkin[8], an open-source distributed tracing system designed to capture timing and dependency data for microservices-based applications.

Zipkin follows the Dapper architecture model (Sigelman et al., 2010) and operates by recording traces, which are composed of one or more spans. Each span represents a single operation or request within a service, and spans are linked to reflect the end-to-end flow of a request through multiple microservices. These traces include metadata such as service name, timestamps, endpoint details, and parent-child relationships between spans.

In this study, each microservice was instrumented with a Zipkin tracer, ensuring that incoming and outgoing HTTP requests were logged as spans and pushed asynchronously to a central Zipkin server. The tracing implementation adhered to the OpenTracing standard, enabling consistent collection across services regardless of language or framework.

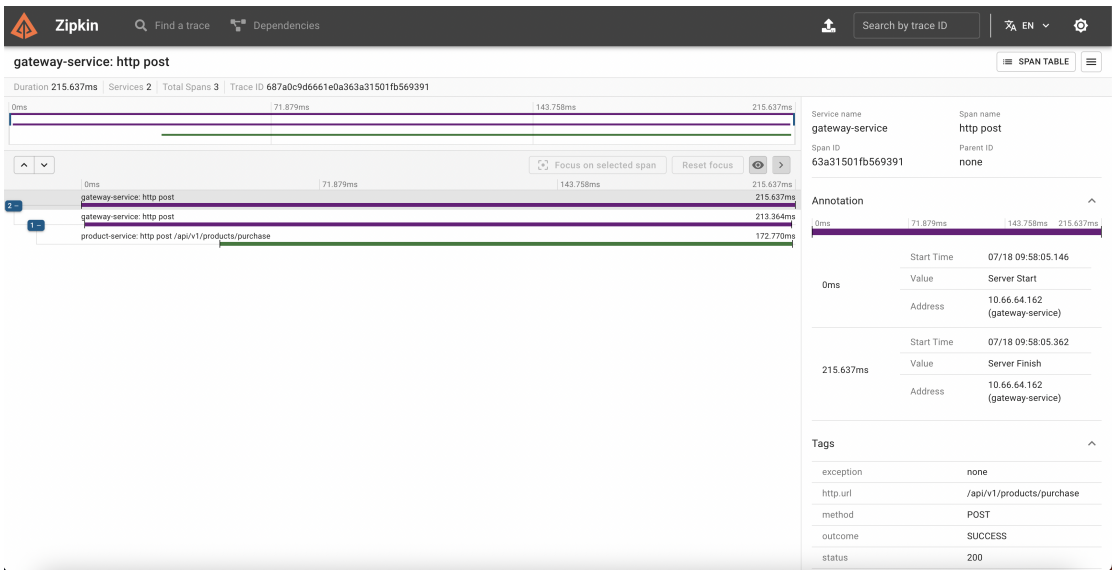


Figure 3.11: Runtime span details of a service invocation

[8]https://zipkin.io/

The collected trace data was exported via Zipkin's RESTful API. These runtime traces were then parsed and transformed into a canonical system model, where:

- Services were modeled as nodes
- Inter-service calls were modeled as directed edges (dependencies)
- Operation frequencies and response patterns were derived from trace sequences.

Instrumentation coverage was verified by manually invoking all public-facing API endpoints of each microservice, ensuring that all critical execution paths were exercised during the trace collection window. Potential limitations of this approach include partial visibility if certain services were not invoked during the capture period or if asynchronous/background tasks were not traceable.

3.7 Reverse Engineering Artifacts

This section describes the reverse-engineering process used to generate a structured JSON input model for the MicroValid static analysis tool. Among the four case studies analysed in this research (see Chapter 4), the Spring PetClinic (Case Study 1) is presented here as an example.

The process of creating the input for MicroValid involved four distinct stages, moving from a high-level architectural overview to a granular definition of data entities and their interrelationships. This approach ensures a comprehensive and accurate representation of the PetClinic application, suitable for subsequent automated analysis.

Step 1: System Analysis and Service Identification

The foundational step involved a high-level analysis of the Spring PetClinic application to understand its domain and identify the boundaries of its constituent microservices. The primary objective was to analyze and document the system's logical service components as defined by its architects. This aligns with the principles of Domain-Driven Design (DDD), which advocates organizing software around business domains, or 'Bounded Contexts' (Evans, 2002).

Through an examination of the project's documentation and repository structure, five services were identified. Infrastructure components such as the config-server and discovery-server were deliberately excluded, as they do not contain domain-specific business logic. The identified services are:

- Customers Service: Manages all data related to pet owners and their pets.
- Vets Service: Manages information about veterinarians and their specialities.
- Visits Service: Responsible for recording and retrieving visit information for pets.
- API Gateway: Acts as a single entry point, routing client requests and aggregating responses from the appropriate downstream services.

This initial identification formed the basis of the `services` array within the JSON artifact.

Step 2: Nanoentity Extraction from Service Domains

Following the high-level service identification, a granular analysis of each microservice's source code was performed to extract its core data structures, or "nanoentities." This concept, introduced by Gysel et al. (2016), represents the most fundamental, indivisible data elements in a system. This involved a static analysis of the domain model classes within each service to identify their persistent fields.

For each service, the Java domain classes (e.g., `Owner`, `Pet`, `Vet`, `Visit`) were examined, and their persistent fields were catalogued. These attributes were then documented as a list of nanoentities under their corresponding service in the JSON model. For example, analysis of the `customers-service` yielded nanoentities such as `Owner.id`, `Owner.firstName`, `Pet.name`, and `PetType.name`. This bottom-up, code-driven analysis ensures that the resulting model is grounded in the actual data implementation of the system, a technique central to many reverse engineering processes.

Entities

1. Owner
    - Nanoentities: ownerId, firstName, lastName, address, city, telephone, pets
2. Pet
    - Nanoentities: petId, name, birthDate, type
3. PetType
    - Nanoentities: petTypeId, name
4. Vet

- Nanoentities: vetId, firstName, lastName, specialties

5. Specialty
    - Nanoentities: specialtyId, name
6. Visit
    - Nanoentities: visitId, date, description, petId

Step 3: Mapping Use Case Responsibilities

To capture the functional dimension of the architecture, the next step was to identify the specific business operations, or use cases, handled by each microservice. This was accomplished by analyzing the exposed public APIs, primarily defined through the REST controllers of each service. This technique of interpreting interface definitions as indicators of service responsibility is widely adopted in microservice identification methodologies.

Each REST endpoint and its corresponding HTTP method in the controller layer was mapped to a distinct use case, represented by its method name. These business responsibilities were formally recorded under the `useCaseResponsibility` structure in the input JSON. For example:

- `customers-service` was linked to operations like "createOwner", "findOwner", "updateOwner", and "getPetTypes".
- `vets-service` supported the "findAllVets" operation, indicating its role in providing veterinary-related data.
- `visits-service` included CRUD operations for managing pet visit records, such as "create" and "read".
- The `api-gateway` aggregated calls to fulfill the "getOwnerDetails" use case, acting as a façade that orchestrates data retrieval from underlying services.
- The `genai-service` exposed a primary "chat" endpoint and defined internal functions like "listOwners" that are callable by the AI model.

This approach enabled a clear delineation of functional boundaries, supporting a coherent mapping of responsibilities to service units within the architecture.

Step 4: Mapping Interservice Relations and Shared Nanoentities

The final step in the reverse engineering process was to model the communication patterns and data dependencies between services. This involved examining how services interact to fulfill composite operations. This method of dependency analysis is foundational to understanding the structural coupling in a software architecture.

Identify Orchestration Layers The analysis began by identifying the central points of interaction. In this architecture, both the `api-gateway` (for the UI) and the `genai-service` (for the chatbot) act as orchestrators, making calls to downstream services.

Trace Synchronous API Calls The logic within these orchestrators was analyzed to trace the direct, synchronous RESTful API calls to the backend services. For each interaction, the target service and the data being exchanged were documented.

Analysis of the ApiGatewayController and its service clients (CustomersServiceClient, VisitsServiceClient) revealed that to fulfill the getOwnerDetails use case, the gateway first calls the customers-service to retrieve owner and pet data, and then calls the visits-service to fetch the visit history for those pets.

Model the Relations Each identified interaction was formally documented in the `relations` array of the MicroValid JSON model. The `direction` property was defined as `INCOMING` relative to the consuming service (`serviceA`). The `sharedEntities` were derived from the Data Transfer Objects (DTOs) used in the API calls. For every relation, the following were captured:

- The consuming service (`serviceA`, e.g., `api-gateway`).
- The providing service (`serviceB`, e.g., `customers-service`).
- The direction of the data flow (`INCOMING` to `serviceA`).
- A list of the `sharedEntities` (nanoentities) that constitute the data contract for that interaction.

The complete JSON input artifact for the Spring PetClinic, generated through the reverse engineering process described above, is provided in Appendix A. This artifact serves as the final structured representation of the system and was used as input to the MicroValid static analysis tool.

3.8 Summary

This chapter outlines the comprehensive methodology used to empirically compare static and dynamic metric tools for evaluating microservice quality. It begins with detailed instructions for setting up the two primary analysis frameworks, MicroValid for static analysis and Bogner et al.'s runtime tool for dynamic analysis, including the configuration of the necessary environment with Docker and Zipkin. The chapter then describes the process of justifying the study's focus on coupling and cohesion through a literature-driven visual mapping of code smells to software metrics. It further details about the systematic reverse-engineering process to create static input models for MicroValid, and the instrumentation of applications with Zipkin to collect live runtime trace data.

Chapter 4

Case Studies

This chapter presents the case studies undertaken to evaluate and compare the performance of static and dynamic metric tools in assessing coupling and cohesion within microservice architectures. The primary aim is to apply MicroValid (static analysis) and Bogner et al., runtime metric tool (dynamic analysis) to identical microservice systems and analyse the consistency, divergence, and complementary value of their outputs.

The chapter begins by introducing the selected sample applications, describing their domain, architecture, and relevance to the research. It then details the experimental procedures applied to each system, including environment configuration, metric collection processes, and analysis workflows. The chapter concludes with a cross-case correlation analysis, a review of the research questions, a discussion of threats to validity, and a concise summary of key findings.

4.1 Overview of Sample Applications

The selection criteria for these application is that the projects should be publicly accessible and is developed using microservice architecture and in addtion to this it should have an integration with Zipkin for distributed tracing. This was essential for collecting the runtime data required by the dynamic analysis tool. This process resulted in the selection of four robust, open-source projects: Spring Petclinic, E-Commerce Backend Platform, Ramanujan, and the Spring Microservices Bookstore.

4.1.1 Spring Petclinic

The Spring Petclinic microservices project is a distributed version of the classic Spring Petclinic sample application, showcasing a modern microservices architecture. This version is built using **Spring Boot** and **Spring Cloud**, demonstrating

how a monolithic application can be decomposed into a set of collaborating services.

The source code is publicly available on GitHub at: `https://github.com/spring-petclinic/spring-petclinic-microservices`

The application is composed of several microservices, each serving a dedicated purpose within the system (Table 4.1):

Table 4.1: Details of Microservices in the Spring Petclinic System

Microservice	**Description**
Admin Server	Provides a dashboard for monitoring and managing the microservices.
API Gateway	Acts as a single entry point for all client requests, routing them to the appropriate services.
Config Server	Centralized configuration for all microservices.
Customers Service	Manages pet owner information.
Discovery Server	A Eureka-based service registry that enables service discovery.
Vets Service	Manages information about veterinarians.
Visits Service	Handles pet visit records.

Through an examination of the project's documentation and repository structure, four services were identified (API Gateway, Customers Service, Vets Service, Visits Service). The services such as "Admin Server", "Config Server", "Discovery Server" are excluded as they are infrastructure components and do not contain domain-specific business logic.

Key features of the application include:

- **Service Discovery**: Enabled through Eureka for dynamic service registration and lookup.
- **Centralized Configuration**: Managed using Spring Cloud Config.
- **API Gateway**: Implemented with Spring Cloud Gateway to handle routing and concerns such as security and monitoring.
- **Distributed Tracing**: Powered by OpenTelemetry and Zipkin for end-to-end traceability.
- **Circuit Breaker**: Implemented using Resilience4j to prevent cascading failures.

The following table (Table 4.2) provides a detailed breakdown of the codebase:

Table 4.2: Project Structure of Spring Petclinic Microservices

Language	**Files**	**Blank**	**Comment**	**Code**
CSS	4	1041	50	8655
XML	16	33	1	2685
Java	62	455	956	1726
Maven	9	83	40	1087
JSON	15	0	0	923
YAML	17	44	13	418
SCSS	4	80	21	412
HTML	11	62	3	324
JavaScript	22	56	15	321
SQL	12	26	0	204
Properties	23	0	20	183
Dockerfile	3	6	4	24
SUM:	**198**	**1886**	**1123**	**16962**

With a total of 16,962 lines of code, the Spring Petclinic microservices project is a mature and detailed example of a cloud-native distributed system. Its modular structure and integration with distributed tracing technologies makes it ideal for this research.

4.1.2 E-Commerce Backend Platform

This project is a comprehensive, fully-completed microservices application designed to showcase modern software architecture and development practices. The application is built using **Spring Boot 3.2.5**, **Java 17**, and **Spring Cloud 2023.0.1**, providing a robust foundation for a distributed system.

The source code is publicly available on GitHub at: `https://github.com/PramithaMJ/fully-completed-microservices-Java-Springboot`

The application is composed of five distinct microservices, each responsible for a specific business capability (Table 4.3):

Table 4.3: Details of Microservices in the E-Commerce Backend Platform

Microservice	**Description**
Gateway Service	API Gateway for routing and cross-cutting concerns.
Customer Service	Manages customer data.
Product Service	Manages product information.
Order Service	Manages orders and their lifecycle.
Payment Service	Handles payment processing.

Key features of the application include:

- **Service Discovery**: All microservices register with the Eureka server for easy discovery.
- **Centralized Configuration**: Configurations are managed centrally using the Spring Cloud Config Server.
- **API Gateway**: Spring Cloud Gateway is used for routing and handling cross-cutting concerns like security, monitoring, and resilience.
- **Distributed Tracing**: Zipkin is used for tracing requests across microservices.
- **Circuit Breaker**: Circuit breaking capabilities provided by Spring Cloud Circuit Breaker.
- **Messaging**: Kafka is used for asynchronous communication between microservices.

The following table (Table 4.4) provides a detailed breakdown of the codebase:

Table 4.4: Project Structure of E-Commerce Backend Platform

Language	**Files**	**Blank**	**Comment**	**Code**
Java	90	391	0	1709
XML	4	18	0	860
Maven	8	36	0	671
Bourne Shell	2	64	125	435
DOS Batch	2	71	0	322
YAML	16	9	0	231
HTML	2	22	0	135
Markdown	8	42	0	124

Continued on next page

Language	Files	Blank	Comment	Code
Text	4	2	0	110
SQL	2	14	6	61
Properties	3	0	0	7
SUM:	**141**	**669**	**131**	**4665**

This project has a total of 4,665 lines of code and serves as a practical example of a microservices architecture. Its well-defined structure and comprehensive feature set make it an ideal case study for understanding and evaluating distributed systems.

4.1.3 Ramanujan

The Ramanujan project is a microblogging system, similar in basic functionality to Twitter, implemented using a microservice architecture. This project, built with Node.js and the Seneca microservice framework. This project was designed to be a learning resource for building and understanding microservice-based systems.

The source code is publicly available on GitHub at: `https://github.com/senecajs/ramanujan`

The application is composed of several microservices (Table 4.5), each with a specific role:

Table 4.5: Ramanujan Microservice Descriptions

Microservice	Description
API	Exposes a RESTful JSON API over HTTP for external clients.
Base	Provides foundational peer-to-peer service discovery using the SWIM protocol, removing the need for a separate service registry.
Entry Cache	Caches microblog entries to reduce latency and load on the entry store.
Entry Store	Manages the persistence of microblog entries.
Fanout	Handles the distribution of new posts to the timelines of followers.
Follow	Manages the relationships between users (i.e., who follows whom).
Front	Provides the server-side rendered user interface.

Continued on next page

Microservice	Description
Home	Manages the user's home page, which displays their timeline.
Index	Provides indexing capabilities for microblog entries.
Mine	Handles the view of a user's own posts.
Post	Manages the creation of new microblog entries.
Repl	A REPL (Read-Eval-Print Loop) service for interacting with the microservice network for debugging.
Reserve	A service whose purpose is not immediately clear from its name, likely for a specific business logic.
Search	Provides search functionality for microblog entries.
Timeline	Manages user timelines, including sharded services for scalability.
Timeline Shard	A sharded version of the timeline service to distribute the load of timeline management.

Key features of the application include:

- **Peer-to-Peer Service Discovery**: Uses a SWIM-based protocol via the `seneca-mesh` library, eliminating the need for a central service registry.
- **Message-Based Communication**: Services communicate via messages, facilitated by the Seneca framework.
- **Development Tooling**: Utilizes `fuge` for managing the lifecycle of microservices in a local development environment.
- **Distributed Tracing**: Integrated with Zipkin for end-to-end message tracing.
- **In-Memory Storage**: Defaults to in-memory storage for rapid development and testing, with the option to add persistence via Seneca plugins.
- **Server-Side Rendering**: The UI is rendered on the server-side, with no client-side JavaScript framework.

The following table (Table 4.6) provides a detailed breakdown of the codebase:

Table 4.6: Ramanujan Codebase Breakdown

Language	Files	Blank	Comment	Code
JavaScript	33	442	185	1563
Markdown	1	168	0	353
YAML	3	37	0	323
make	18	84	0	250
JSON	2	0	0	228
HTML	6	24	0	110
Dockerfile	17	57	0	66
Text	1	54	0	51
CSS	1	9	0	46
Bourne Shell	1	8	1	38
SUM:	**83**	**883**	**186**	**3028**

With a total of 3,028 lines of code, the Ramanujan microservices project is a concise and focused example of a microservice system. Its clear structure and lack of external dependencies (beyond Node.js modules) make it an ideal choice for research.

4.1.4 Spring Microservices Bookstore

The Spring Microservices Bookstore Demo is a project designed to illustrate the patterns and practices used when building and managing microservices with Spring. It showcases a modern microservices architecture with a variety of integrated technologies to build a scalable and maintainable system.

The source code is publicly available on GitHub at: `https://github.com/chris-bailey/spring-microservices-bookstore-demo`

The application is composed of several microservices (Table 4.7), each serving a dedicated purpose within the system:

Table 4.7: Spring Microservices Bookstore Service Descriptions

Microservice	Description
API Gateway	Acts as the entry point to the microservices architecture, routing requests to the appropriate services.
Book Service	Manages book data and interacts with a MongoDB database, using GraphQL for querying.
Author Service	Manages author profiles and interactions using a non-blocking, reactive approach with Spring Webflux.

Continued on next page

Microservice	Description
Order Service	Handles purchase transactions and interacts with a PostgreSQL database.
StockCheck Service	Manages stock checking and interacts with a PostgreSQL database, written in Kotlin.
Message Service	Sends messages through interactions with Kafka.

Key features of the application include:

- **Service Discovery**: Enabled through Eureka for dynamic service registration and lookup.
- **Centralized Configuration**: Managed using Spring Cloud Config.
- **API Gateway**: Implemented with Spring Cloud Gateway to handle routing and concerns such as security and monitoring.
- **Distributed Tracing**: Powered by Micrometer Tracing, Brave, and Zipkin for end-to-end traceability.
- **Circuit Breaker**: Implemented using Resilience4j to prevent cascading failures.
- **Database**: Uses both PostgreSQL and MongoDB to store data.
- **Event-Driven Architecture**: Leverages Apache Kafka for asynchronous communication between services.
- **Containerization**: Docker and Docker Compose for containerizing and orchestrating the application, with Kubernetes support for deployment.
- **Monitoring**: Uses Prometheus and Grafana for monitoring and visualizing metrics.
- **Frontend**: A Next.js and React-based frontend with TypeScript and Tailwind CSS.

The following table (Table 4.8) provides a detailed breakdown of the codebase:

Table 4.8: Spring Microservices Bookstore Codebase Breakdown

Language	**Files**	**Blank**	**Comment**	**Code**
JSON	6	0	0	8938
YAML	50	87	21	1702
Java	41	212	19	1002
Maven	9	35	7	708
Markdown	1	137	0	417
TypeScript	4	41	2	388
Kotlin	8	36	1	173
CSS	1	4	0	29
GraphQL	1	3	0	18
SQL	5	0	0	16
Dockerfile	1	7	8	8
JavaScript	2	2	2	8
SUM:	**129**	**564**	**60**	**13407**

The Spring Microservices Bookstore has a total of 13,407 lines of code with integrations with various modern technologies, including distributed tracing enabled by Zipkin, making it an ideal choice for this research, as this feature was essential for collecting the runtime data required for dynamic analysis.

4.1.5 Summary of Case Studies

Table 4.9 shows the number of microservices per case study. These services were then analysed in detail under Section 4.2.

Table 4.9: Summary of Microservices per Case Study

Case Study	**Number of Microservices**
Spring Petclinic	4
E-Commerce Backend Platform	5
Ramanujan	14
Spring Microservices Bookstore	6

4.1.6 Justification for Case Study Selection

Spring Petclinic is frequently used to evaluate structural qualities such as coupling and cohesion, and several peer-reviewed studies have shown that its official

microservice decomposition exhibits moderate coupling and low cohesion, which actually makes it useful for benchmarking architectural analysis tools. The study (Zaverdehi, 2024) used Average Coupling, Average Cohesion, and Independence of Functionality (IFN), as well as SOA-derived metrics such as CBM, AIS, ADS, SIDC, and SIUC. Petclinic has also been evaluated using response time, throughput, and CVSS scores in performance and security analyses.

Ramanujan, on the other hand, is the reference implementation used in the original Bogner et al. paper that introduced the runtime metric tool used in this thesis. It has been shown to exhibit loosely coupled, message-based communication and peer-to-peer service discovery, and the original study validated the correctness of the tool's runtime coupling and cohesion metrics directly on this system.

The E-Commerce Backend and Spring Bookstore systems have not yet been assessed in academic literature, so there is no external quality baseline available. Their value in this research lies in the fact that they satisfy all technical prerequisites (publicly available, actual MSA, Zipkin integration). By applying the tools to these systems, the thesis is able to evaluate static and dynamic metrics on systems that more closely resemble typical industry applications, rather than curated academic benchmarks. In this sense, the thesis contributes new empirical quality assessments for these two systems.

4.2 Experiments

The detailed output and input files used in all the experiments can be accessed in my GitHub repository: https://github.com/adarshajit/msc-thesis

4.2.1 Objective

The objective of these experiments was to empirically evaluate how well static architectural metrics (as computed by MicroValid) align with runtime metrics (as produced by Bogner et al., tool). Specifically, the experiments set out to answer:

- Whether static metrics (coupling and cohesion) meaningfully predict analogous runtime behavior.
- Under what conditions static analysis can serve as a reliable proxy for architectural quality observed at runtime.

4.2.2 Environment

The local machine used for this setup was configured with the following technical specifications:

- Operating System: macOS (Apple M2 Silicon)
- Processor: Apple M2, 8-core (4 performance + 4 efficiency)
- Development Environment: Visual Studio Code (VSCode) & IntelliJ IDEA 2025.1.3 (Ultimate Edition)
- Containerization: Docker/Docker Compose for Zipkin and service orchestration
- Other Tools: Git, SDKMAN!, Java 17+

4.2.3 Experiment 1: Spring PetClinic

Prerequisites

Before running the experiment, ensure the following conditions are met:

The Spring Petclinic microservices application is fully up and running. You can verify the status of the services by checking the Eureka server dashboard (Figure 4.1), which should show all registered services, or by listing the running Docker containers using the command `docker ps`.

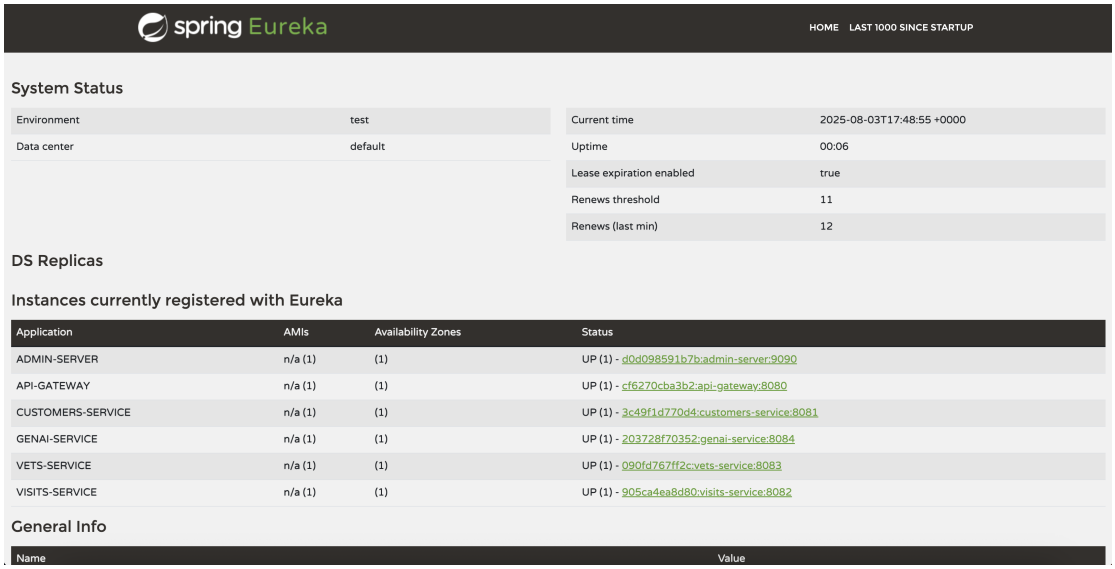


Figure 4.1: Eureka Server indicating all the running services for Spring Pet Clinic

The Zipkin instance for distributed tracing is active (Figure 4.2). The application will not function correctly without this, as it is a key component for collecting the data required by the runtime analysis tool.

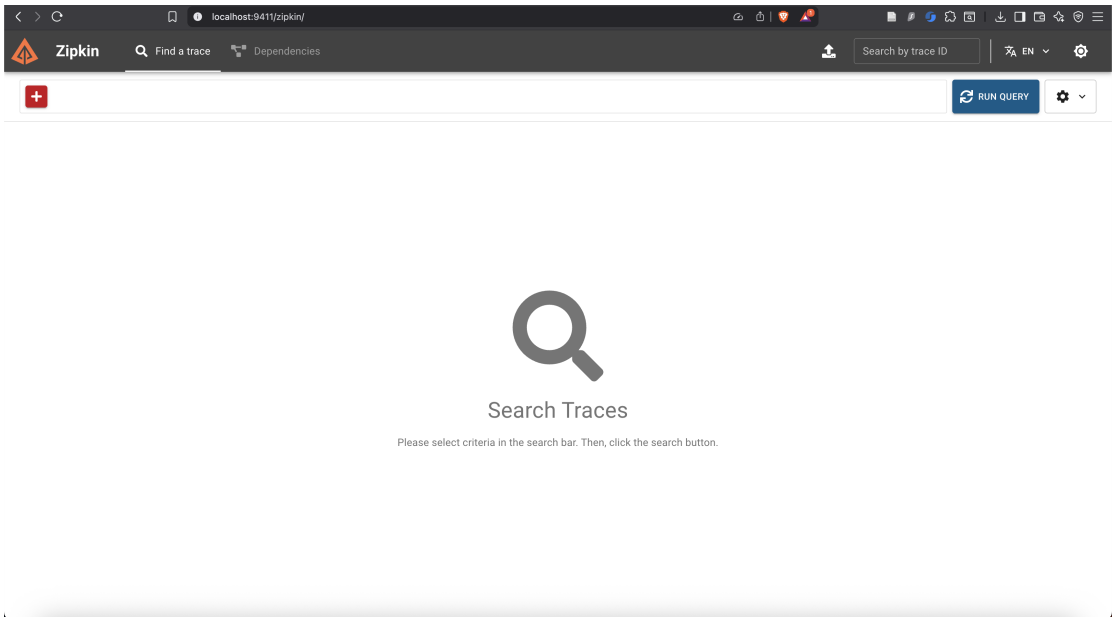


Figure 4.2: Zipkin Dashboard running on the default PORT: 9411

Results

MicroValid Results

Table 4.10 shows the detailed metric values obtained from MicroValid:

Table 4.10: MicroValid Results for Spring Petclinic Microservices

Spring Petclinic Microservices			
Quality Attribute	**MicroValid Test**	**Score**	**Verdict**
Coupling	DEPENDENCIES_COMPOSITION_TEST	7.3	Acceptable
	SCC_TEST	10	Good
Cohesion	ENTITIES_COMPOSITION_TEST	1	Failed
	RESPONSIBILITIES_COMPOSITION_TEST	0.3	Failed
	RELATIONS_COMPOSITION_TEST	6.7	Acceptable
	SEMANTIC_SIMILARITY_TEST	6	Acceptable

MicroValid Score Tiers (Cojocaru et al., 2019):

- **Good:** $(7.5, 10.0]$
- **Acceptable:** $(5.0, 7.5]$
- **Failed:** $[0.0, 5.0]$

The total MicroValid Coupling and Cohesion scores for Spring PetClinic are as follows:

- Total Coupling Score: 8.65 (Good)
- Total Cohesion Score: 3.5 (Failed)

Bogner et al., Runtime Metric Tool Results

The below Table 4.11 and Table 4.12 shows the detailed metric values obtained from Bogner Runtime Metric Tool.

Bogner et al., Metric Results						
Quality Attribute		**Spring PetClinic Microservices**				**Average**
		visits-service	**api-gateway**	**customers-service**	**vets-service**	
Coupling	AIS	1	0	1	1	**0.75**
	ADS	0	0	0	0	**0**
	ACS	0	0	0	0	**0**
	MAIDS	0.75				**0.75**
	MACS	1.5				**1.5**
	SIY	0				**0**
Cohesion	SIUC	33%	0	100%	0	**33.25%**
	SIDC	0	0	0	0	**0%**
	TSIC	16.5%	0	50%	0	**16.63%**
	IUAM	0.12				**0.12**

Table 4.11: Bogner et al., Metric results for Spring Petclinic.

Table 4.12: Normalised Bogner et al., Metric results for Spring Petclinic.

Normalised Bogner et al., Results					
Quality Attribute	**Metric**	**Raw Value**	**Max Value (n=4)**	**Score (0–10)**	**Verdict**
Coupling	AIS (avg)	0.75	3	7.5	Acceptable
	ADS (avg)	0	3	10.0	Good
	ACS (avg)	0	9	10.0	Good
	MAIDS	0.75	3	7.5	Acceptable
	MACS	1.5	6	7.5	Acceptable
	SIY	0	6	10.0	Good
	Coupling Composite	—		**8.75**	**Good**
Cohesion	TSIC (avg)	0.16625	1	1.7	Failed
	IUAM	0.12	1	1.2	Failed
	Cohesion Composite	—		**1.4**	**Failed**

Glossary - n : Number of Services, AIS: Absolute Importance of the Service, ADS: Absolute Dependence of the Service, ACS: Absolute Criticality of the Service, MAIDS: Mean Absolute Importance/Dependence in the System, MACS: Mean Absolute Coupling in the System, SIY: Services Interdependence in the System, TSIC: Total Service Interface Cohesion, IUAM: Inverse of Average Number of Used Message

Analysis

MicroValid Metric Suite: MicroValid Metric Suite: Based on the MicroValid test results, the coupling scores are generally strong, with clear explanations for each outcome. The perfect SCC TEST score of 10 is expected, as there are no circular dependencies. The DEPENDENCIES COMPOSITION TEST score of 7.3 is rated "Acceptable." Three services, customers, vets, and visits have zero outward dependencies, making them independent providers of business logic. The api-gateway, however, has two outward dependencies, serving as the single point of entry and coordination. While this distribution is not perfectly balanced, it reflects an intentional architectural decision, which the "Acceptable" rating appropriately captures.

In contrast, the ENTITIES COMPOSITION TEST (Score: 1.0 – Failed) highlights an imbalanced distribution of entities ("customers-service" has 3 entities, "vets-service" has 2 entities, "visits-service" has 1 entities, "api-gateway" has 0 entities), with the customers service emerging as noticeably larger than the others. The RESPONSIBILITIES COMPOSITION TEST (Score: 0.3 – Failed) indicating that there high variation in responsibilities: the customers service handles 8 use cases, while other services manage only 1 or 2.

Normalisation and Scoring Method for Bogner et al., Results

The Bogner et al., runtime metrics are reported in raw measurement units (e.g., number of dependencies, percentages, or proportions), meaning they differ substantially in both scale and semantic meaning (e.g., “number of incoming calls” vs. “proportion of cyclic dependencies”). As a result, they are not on a common scale and cannot be directly compared either within the same quality attribute category or against MicroValid’s static analysis scores. This makes a scale-independent statistical measure such as the coefficient of variation (used in MicroValid) unsuitable, since it assumes that all variables are expressed in comparable units and distributions. Applying it to heterogeneously scaled runtime data would produce mathematically valid but non-intuitive values that are difficult to compare across metrics.

To enable meaningful cross-metric comparison and produce composite coupling and cohesion scores, a min–max normalisation to a 0–10 scale was applied, using the theoretical maximum possible value for each metric as the upper bound. This approach ensures that all runtime measures contribute proportionally and consistently when aggregated, while preserving interpretability (Table 4.12)

For coupling metrics (lower is better), scores were inverted so that higher scores

represent better quality:

$$\text{Score}_{(0\text{˜}10)} = \left(1 - \frac{\text{Raw Value}}{\text{Max Value}}\right) \times 10$$

For cohesion metrics (higher is better), scores were scaled directly:

$$\text{Score}_{(0\text{˜}10)} = \frac{\text{Raw Value}}{\text{Max Value}} \times 10$$

In the Petclinic example:

- **AIS** $= 0.75 \rightarrow \left(\frac{0.75}{3}\right) \times 10 = 7.5$
- **ADS** $= 0 \rightarrow \left(\frac{0}{3}\right) \times 10 = 10.0$
- **SIY** $= 0 \rightarrow \left(1 - \frac{0}{6}\right) \times 10 = 10.0$

This approach ensures that all normalised scores are directly comparable and aligned in direction (higher score = better quality).

For each quality attribute:

- Total Coupling Score = Arithmetic mean of normalised scores for all coupling-related metrics (AIS, ADS, ACS, MAIDS, MACS, SIY).
- Total Cohesion Score = Arithmetic mean of normalised scores for all cohesion-related metrics (TSIC, IUAM).

The total Bogner et al., Coupling and Cohesion scores for Spring PetClinic are as follows:

- Total Coupling Score: 8.75 (Good)
- Total Cohesion Score: 1.4 (Failed)

Bogner et al., Runtime Analysis Tool

The Bogner et al., results for the Spring PetClinic system show that the services behave quite differently in terms of coupling and cohesion. The customers-service has strong cohesion (100%) and only a small amount of coupling, meaning it is well-focused on its own responsibilities and does not depend much on other services. The vets-service shows medium cohesion (50%) with low coupling, which suggests it is partly specialised but still has some overlap or external reliance. The visits-service has weaker cohesion (33%) and higher interaction with others, which means its responsibilities are more scattered and changes could spread more easily across the system. The api-gateway, on the other hand, shows almost no

cohesion or coupling, which fits its role as a lightweight entry point rather than a provider of business logic. These results indicate that while some services are well-structured and independent, others (especially visits-service) may reduce system quality because of weaker cohesion and higher coupling.

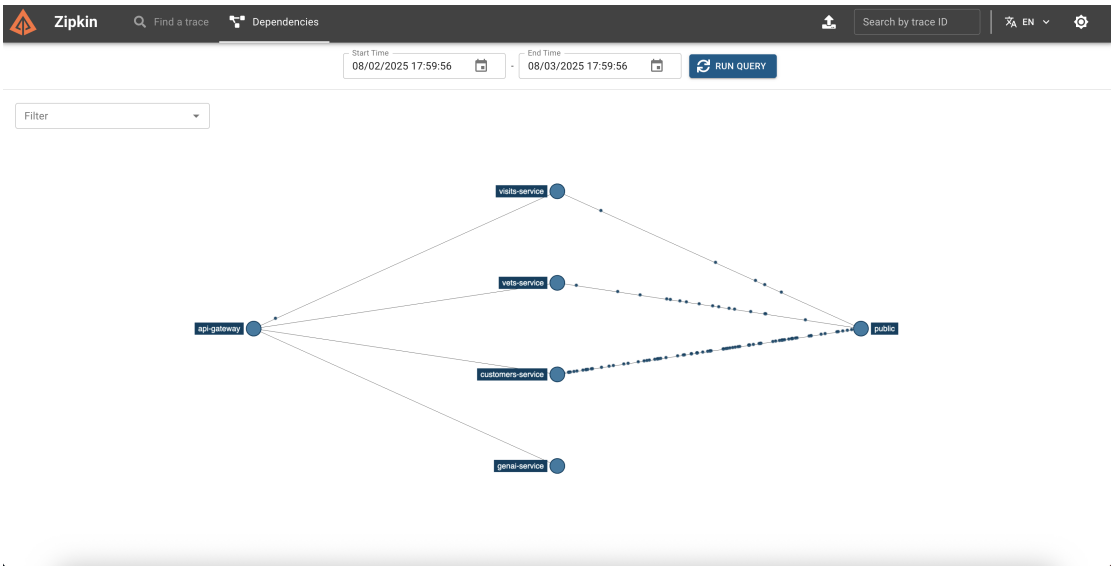


Figure 4.3: Zipkin Dependency graph for Spring PetClinic

While MicroValid's summary score suggests "acceptable" cohesion, Bogner et al., runtime tool is more granular and often poor scores for SIDC and SIUC reveal deep inconsistencies in the functional and data cohesion of the services at runtime.

4.2.4 Qualitative Evaluation by Subject Expert for Spring PetClinic

MicroValid Analysis Tool

There is a significant disagreement here. The qualitative evaluation views the customers-service encapsulating a specific business domain as a sign of good, focused design. In contrast, MicroValid, which uses the coefficient of variation to assess the balance of a system, flags this as a major flaw. The failed scores for the ENTITIES COMPOSITION TEST (1.0) and RESPONSIBILITIES COMPOSITION TEST (0.3) are precisely because the customers-service is disproportionately larger than its peers. What this assessment identifies as a strength, MicroValid marks as a failure in decomposition balance.

Bogner et al. Runtime Analysis Tool

While the two analyses agreed on the customers-service, the overall positive qualitative assessment overlooks the weakness identified by Bogner's tool. Bogner et

al. found the visits-service to have weaker cohesion (33%) and higher interaction, suggesting it could reduce system quality. The qualitative opinion makes no mention of such a weak link in the architecture.

Final Verdict

The qualitative opinion aligns more with Bogner et al. because both share a positive assessment of the system's most significant component, the customers-service. However, this high-level, qualitative praise does not capture the nuanced weaknesses that both quantitative tools reveal. The opinion is largely at odds with MicroValid, whose metrics are specifically designed to penalize the exact kind of architectural imbalance perceived as a positive design choice in this evaluation.

4.2.5 Experiment 2: E-Commerce Backend Platform

Prerequisites

Before running the experiment, ensure the following conditions are met: The application is fully up and running, with all microservices registered with the Eureka server (Figure 4.4). You can verify this by checking the Eureka server dashboard or by listing the running Docker containers. All services, including the Discovery Service and Config Server, should be properly registered and communicating.

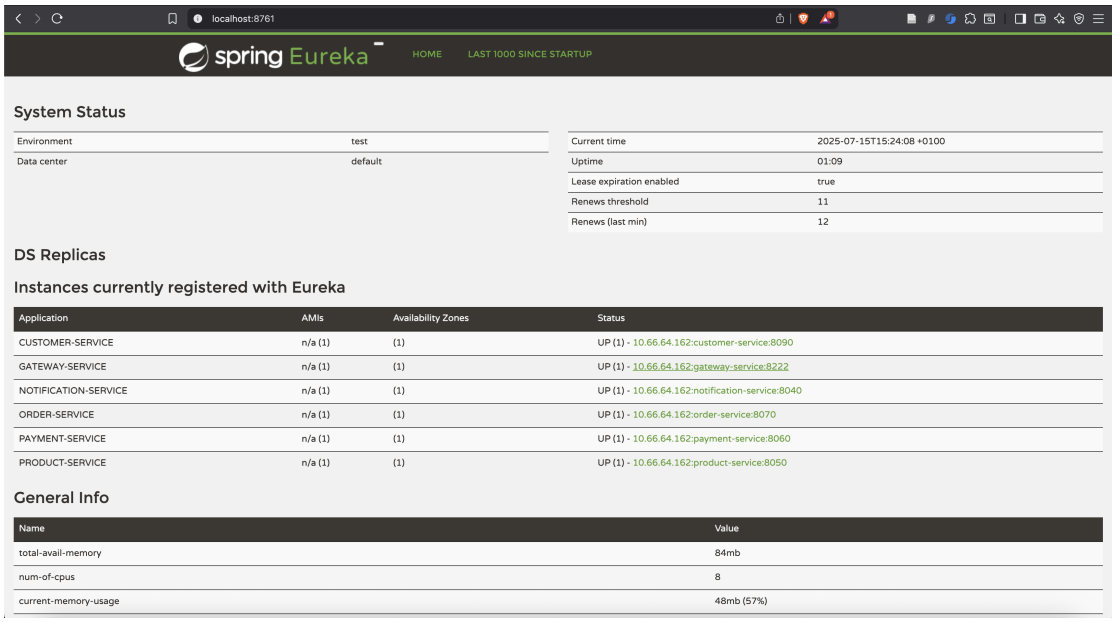


Figure 4.4: Eureka dashboard indicating all the running services for E-Commerce Backend Platform

The Zipkin instance for distributed tracing is active (Figure 4.5), as this is a key component for collecting the data required by the runtime analysis.

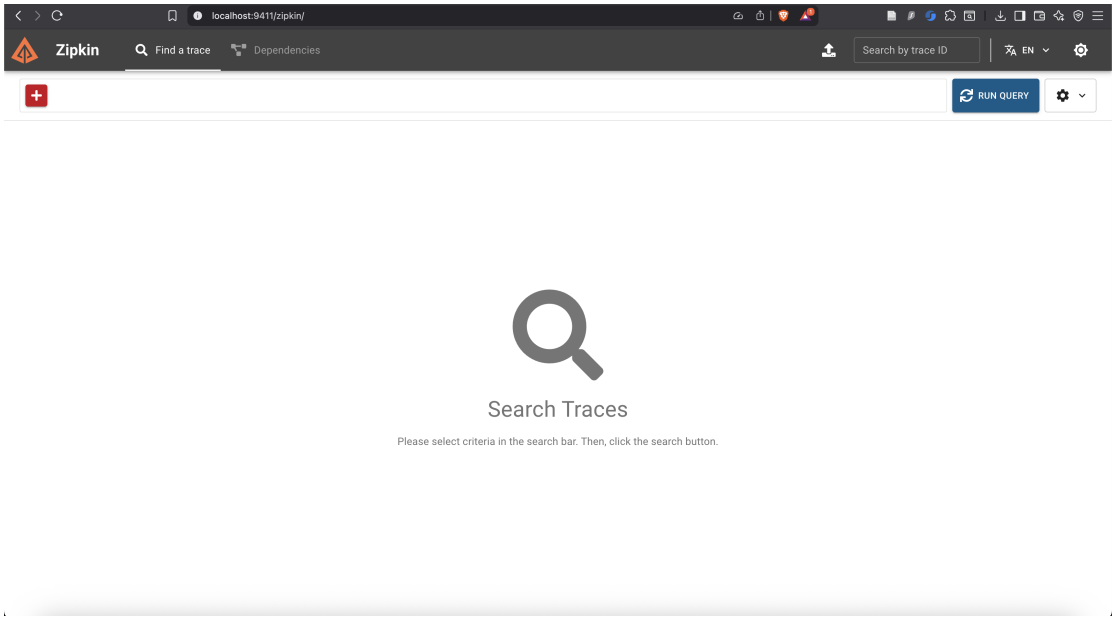


Figure 4.5: Zipkin Dashboard running on the default PORT: 9411

Failure to meet these prerequisites may result in incomplete or erroneous experimental results.

Results

MicroValid Results

Table 4.13 shows the detailed metrics values obtained from MicroValid:

Table 4.13: MicroValid results for E-Commerce Backend Platform

E-Commerce Backend Platform			
Quality Attribute	**MicroValid Test**	**Score**	**Verdict**
Coupling	DEPENDENCIES_COMPOSITION_TEST	4.0	Failed
	SCC_TEST	10.0	Good
Cohesion	ENTITIES_COMPOSITION_TEST	1.8	Failed
	RESPONSIBILITIES_COMPOSITION_TEST	0.0	Failed
	RELATIONS_COMPOSITION_TEST	0.0	Failed
	SEMANTIC_SIMILARITY_TEST	7.8	Good

MicroValid Score Tiers (Cojocaru et al., 2019):

- **Good:** $(7.5, 10.0]$
- **Acceptable:** $(5.0, 7.5]$
- **Failed:** $[0.0, 5.0]$

The total MicroValid Coupling and Cohesion scores for E-Commerce Backend Platform are as follows:

- Total Coupling Score: 7 (Acceptable)
- Total Cohesion Score: 2.4 (Failed)

Bogner et al., Runtime Metric Tool Results

The below Tables 4.14 and 4.15 shows the detailed metric values obtained from Bogner et al., Runtime Metric Tool

Bogner et al., Metric Results							
Quality Attribute		**E-Commerce Backend Platform Microservices**					**Average**
		payment-service	**customer-service**	**gateway-service**	**product-service**	**order-service**	
Coupling	AIS	1	1	1	1	0	**0.8**
	ADS	0	0	0	0	3	**0.6**
	ACS	0	0	0	0	0	**0**
	MAIDS	0.7					**0.7**
	MACS	1.4					**1.4**
	SIY	0					**0**
Cohesion	SIUC	100%	0	0	0	0	**20%**
	SIDC	0	100%	0	0	0	**20%**
	TSIC	50%	50%	0	0	0	**20%**
	IUAM	0.38					**0.38**

Table 4.14: Bogner et al., Metric Results for E-Commerce Backend Platform

Table 4.15: Normalised Bogner et al., Metric Results for E-Commerce Backend Platform

Normalised Bogner Results					
Quality Attribute	**Metric**	**Raw Value**	**Max Value (n=5)**	**Score (0–10)**	**Verdict**
Coupling	AIS (avg)	0.8	4	8	Good
	ADS (avg)	0.6	4	8.5	Good
	ACS (avg)	0	16	10	Good
	MAIDS	0.7	4	8.3	Good
	MACS	1.4	8	8.3	Good
	SIY	0	10	10	Good
	Total Coupling Score	—		**8.83**	**Good**
Cohesion	TSIC (avg)	0.2	1	2	Poor
	IUAM	0.38	1	3.8	Poor
	Total Cohesion Score	—		**2.9**	**Poor**

Glossary - n : Number of Services, AIS: Absolute Importance of the Service, ADS: Absolute Dependence of the Service, ACS: Absolute Criticality of the Service, MAIDS: Mean Absolute Importance/Dependence in the System, MACS: Mean Absolute Coupling in the System, SIY: Services Interdependence in the System, TSIC: Total Service Interface Cohesion, IUAM: Inverse of Average Number of Used Message

Analysis

MicroValid Metric Suite: The MicroValid analysis reveals significant architectural issues with the system's coupling. The DEPENDENCIES COMPOSITION TEST fails with a score of 4.0, which the details attribute to a severe structural imbalance. The order service acts as a major dependency hub with 5 outward dependencies, while most other services have none. This points to a problematic design where a single service is overly responsible for orchestrating the system, creating tight coupling.

On the aspect of cohesion, with three of the four tests failing. The RESPONSIBILITIES COMPOSITION TEST and RELATIONS COMPOSITION TEST both scored zero. This is due to the imbalance created due to core functionalities like the findById use case which was duplicated across three separate services, and key data models are duplicated in the communication paths. This suggests the service boundaries are critically blurred and do not adhere to the single responsibility principle.

Normalisation and Scoring Method for Bogner et al., Results

The same steps used to normalise the Bogner et al., results for other studies were used here as well.

The total Bogner et al., Coupling and Cohesion scores for E-Commerce Backend Platform are as follows:

- Total Coupling Score: 8.83 (Good)
- Total Cohesion Score: 2.9 (Failed)

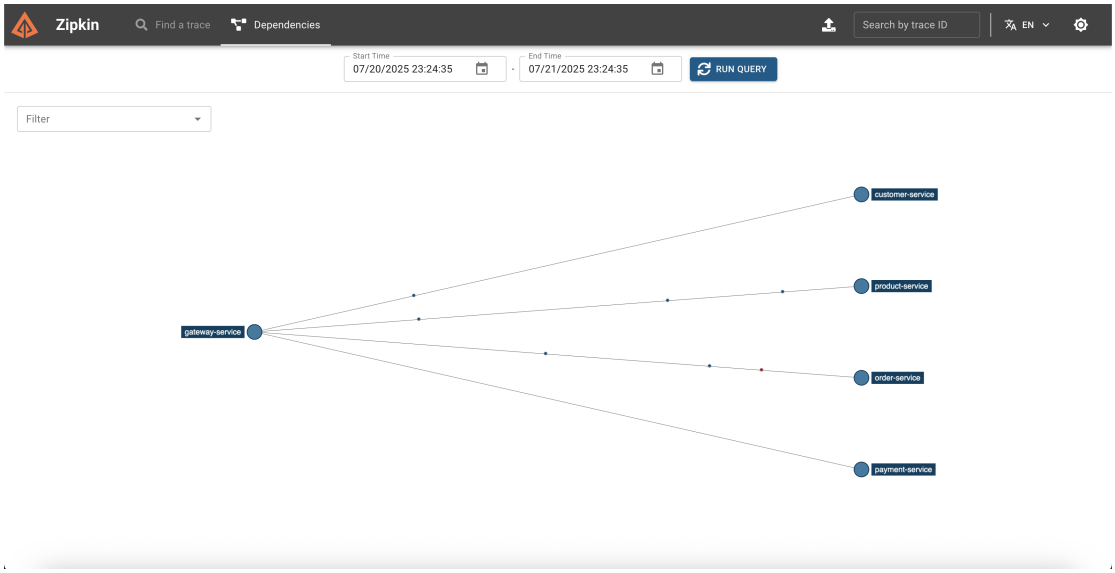


Figure 4.6: Zipkin Dependency graph for E-Commerce Backend Platform

The results indicate inter-service coupling, the order-service stands out as a central dependency hub. It makes three outgoing calls to other services (ADS=3), while all other components make none. Cohesion is also inconsistent as The payment-service shows perfect functional cohesion, meaning its operations are always used together by clients (SIUC=100%), while the customer-service demonstrates perfect data cohesion, as its operations all relate to a common set of data types (SIDC=100%). However, neither service is strong in both aspects, and the remaining services lack any measurable cohesion (TSIC=0), suggesting their responsibilities are poorly focused.

4.2.6 Experiment 3: Ramanujan

Prerequisites

Before running the experiment, ensure the following conditions are met:

Start by making sure the application is running correctly in local (Figure 4.7). All the instructions to set it up was in the README.md [1] of the github repo of this codebase.

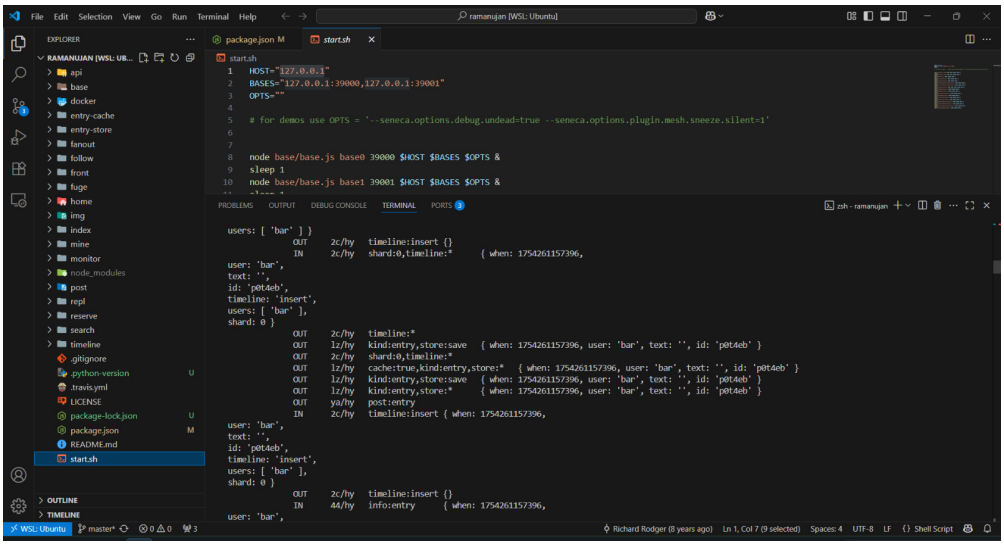


Figure 4.7: Ramanujan running successfully in local

The Zipkin instance for distributed tracing is active (Figure 4.8). The application uses Zipkin for end-to-end message tracing, and its proper functioning is crucial for data collection.

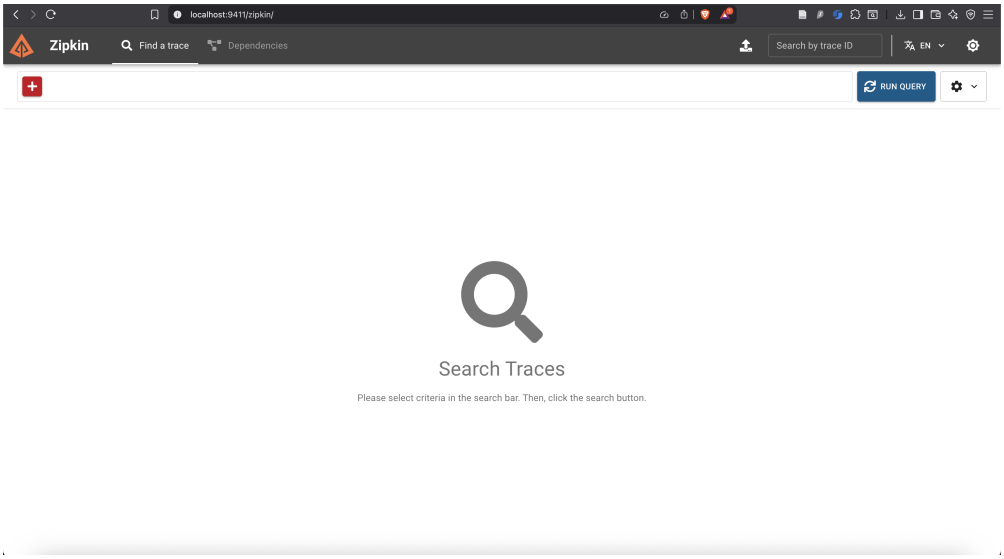


Figure 4.8: Zipkin Dashboard running on the default PORT: 9411

[1] https://github.com/senecajs/ramanujan/blob/master/README.md

Results

MicroValid Results

The following table represents the MicroValid results for Ramunjan codebase (Table: 4.16)

Table 4.16: MicroValid results for Ramanujan

Ramanujan			
Quality Attribute	**MicroValid Test**	**Score**	**Verdict**
Coupling	SCC_TEST	8.5	Good
	DEPENDENCIES_COMPOSITION_TEST	3.3	Failed
Cohesion	ENTITIES_COMPOSITION_TEST	0.8	Failed
	RESPONSIBILITIES_COMPOSITION_TEST	0	Failed
	RELATIONS_COMPOSITION_TEST	0	Failed
	SEMANTIC_SIMILARITY_TEST	10	Good

MicroValid Score Tiers (Cojocaru et al., 2019):

- **Good:** $(7.5, 10.0]$
- **Acceptable:** $(5.0, 7.5]$
- **Failed:** $[0.0, 5.0]$

The total MicroValid Coupling and Cohesion scores for Ramanujan are as follows:

- Total Coupling Score: 5.9 (Acceptable)
- Total Cohesion Score: 2.7 (Failed)

Bogner et al., Runtime Metric Tool Results

The below Table 4.17 and Table 4.18 shows the detailed metric values obtained from Bogner et al., Runtime Metric Tool.

Bogner et al., Metric Results																
Quality Attribute		**Ramanujan Microservices**														**Average**
		mine	**fanout**	**entry-cache**	**index**	**follow**	**timeline0**	**home**	**timeline1**	**search**	**post**	**reserve**	**api**	**timeline-shard**	**entry-store**	
Coupling	AIS	0	1	3	2	5	3	0	3	0	1	3	0	3	3	**1.92**
	ADS	2	1	0	0	6	2	3	2	2	4	0	2	0	3	**1.92**
	ACS	0	1	0	0	30	6	0	6	0	4	0	0	0	9	**4.0**
	MAIDS	1.93														**1.93**
	MACS	3.86														**3.86**
	SIY	2														**2**
Cohesion	SIUC	nan	100%	67%	50%	50%	50%	nan	50%	nan	100%	100%	nan	75%	50%	**49.42%**
	SIDC	nan	0%	67%	67%	100%	100%	nan	100%	nan	100%	100%	nan	100%	100%	**59.57%**
	TSIC	nan	50%	67%	67%	75%	75%	nan	75%	nan	100%	100%	nan	87.50%	75%	**55.1%**
	IUAM	0.61														**0.61**

Table 4.17: Bogner et al., Metrics for Ramanujan

Table 4.18: Normalised Bogner et al., Results for Ramanujan

Normalised Bogner et al., Results					
Quality Attribute	**Metric**	**Raw Value**	**Max Value (n=14)**	**Score (0–10)**	**Verdict**
Coupling	AIS (avg)	1.92	13	8.5	Good
	ADS (avg)	1.92	13	8.5	Good
	ACS (avg)	4	169	9.8	Good
	MAIDS	1.93	13	8.5	Good
	MACS	3.86	26	8.5	Good
	SIY	2	91	9.8	Good
	Total Coupling Score	—		**8.9**	**Good**
Cohesion	TSIC (avg)	55.1	1	5.5	Acceptable
	IUAM	0.61	1	6.1	Acceptable
	Total Cohesion Score	—		**5.8**	**Acceptable**

Glossary - n : Number of Services, AIS: Absolute Importance of the Service, ADS: Absolute Dependence of the Service, ACS: Absolute Criticality of the Service, MAIDS: Mean Absolute Importance/Dependence in the System, MACS: Mean Absolute Coupling in the System, SIY: Services Interdependence in the System, TSIC: Total Service Interface Cohesion, IUAM: Inverse of Average Number of Used Message

MicroValid Analysis

The MicroValid results for the Ramanujan application reveal a problematic coupling structure. While the SCC TEST score of 8.5 is rated as "Good," the details uncover a critical architectural flaw: a strongly connected component involving the follow, timeline-shard, and entry-store services, indicating a cyclic dependency that requires these services to be merged. This is compounded by a "Failed" DEPENDENCIES COMPOSITION TEST (3.3), caused by a severe imbalance where multiple services such as "front", "post", and "follow" act as dependency hubs, creating a tangled and "chatty" communication network.

The system's cohesion is also flawed, The RESPONSIBILITIES COMPOSITION TEST and RELATIONS COMPOSITION TEST both scored zero, a result of widespread duplication of core business logic (e.g., listEntries, saveEntry) and data models across different services. This fundamental blurring of service boundaries is supported by a "Failed" ENTITIES COMPOSITION TEST (score: 0.8) due to a sparse and imbalanced entity distribution. The perfect score for SEMANTIC SIMILARITY TEST (10) stands in stark contrast to these structural failures, suggesting that while the service names are conceptually coherent, they hide the underlying issues with respect to incorrect service decomposition.

Normalisation and Scoring Method for Bogner et al., Results

The same steps used to normalise the Bogner et al., results for Spring PetClinic case study was done here as well, the total Bogner et al., Coupling and Cohesion scores for Ramanujan are as follows:

- Total Coupling Score: 8.9 (Good)
- Total Cohesion Score: 5.8 (Acceptable)

Bogner et al., Runtime Metrics Analysis

The Bogner et al., metrics, derived from runtime data, provide a more detailed and positive view of the Ramanujan system's quality, though they also highlight critical architectural flaws. For coupling, the presence of issues is confirmed, as the SIY (Services Interdependence in the System) score is 2, explicitly identifying two pairs of services with problematic bidirectional dependencies. The ADS (Absolute Dependence of the Service) scores vary, with api and follow having a score of 3 and order-service having a score of 2, indicating that these are the most dependent components in the system. The ACS (Absolute Criticality of the Service) metric pinpoints follow (30) and entry-store (9) as the most critical services. In contrast to MicroValid, the cohesion scores are generally good, with TSIC (Total Service

Interface Cohesion) scores ranging from 50% to 87.50% across most services. The system also shows a good overall interface usage aggregation (IUAM) score of 0.61.

4.2.7 Experiment 4: Spring Microservices Bookstore

Prerequisites

Before running the experiment, ensure the following conditions are met: The application is fully up and running, with all microservices registered with the Eureka server (Figure 4.9). You can verify this by checking the Eureka server dashboard or by listing the running Docker containers.

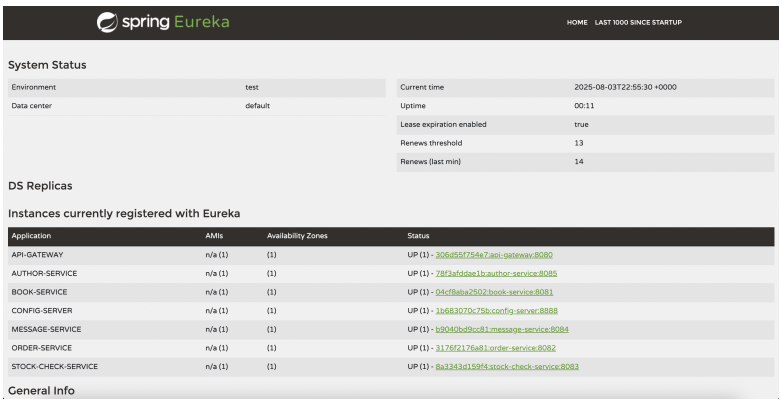


Figure 4.9: Eureka Server indicating all the running services for Spring Microservices Bookstore

The Zipkin instance for distributed tracing is active (Figure 4.10), as this is a key component for collecting the data required by the runtime analysis tool.

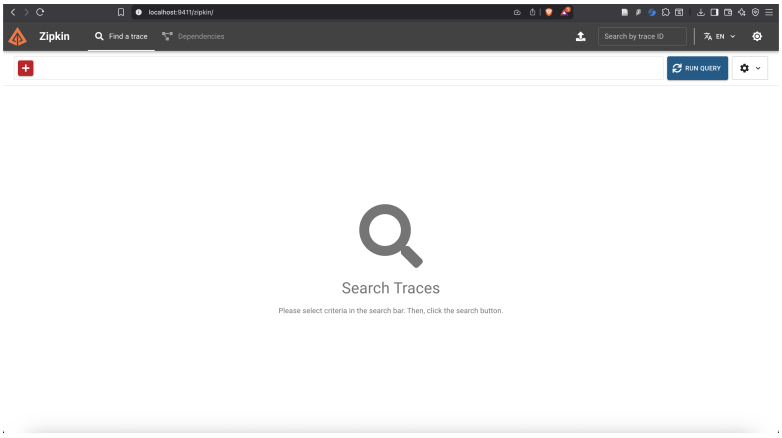


Figure 4.10: Zipkin Dashboard running on the default PORT: 9411

Results

MicroValid Results

Table 4.19 shows the detailed metrics obtained from both approaches:

Table 4.19: MicroValid Results for Spring Microservices Bookstore

Spring Microservices Bookstore		
Quality Attribute	**MicroValid Test**	**Score**
Coupling	SCC_TEST	10
	DEPENDENCIES_COMPOSITION_TEST	2.2
Cohesion	ENTITIES_COMPOSITION_TEST	2
	RESPONSIBILITIES_COMPOSITION_TEST	4.6
	RELATIONS_COMPOSITION_TEST	10
	SEMANTIC_SIMILARITY_TEST	6.6

MicroValid Score Tiers (Cojocaru et al., 2019):

- **Good:** $(7.5, 10.0]$
- **Acceptable:** $(5.0, 7.5]$
- **Failed:** $[0.0, 5.0]$

The total MicroValid Coupling and Cohesion scores for Spring Microservices Bookstore are as follows:

- Total Coupling Score: 6.1 (Acceptable)
- Total Cohesion Score: 5.8 (Failed)

Bogner et al., Runtime Metric Tool Results

The below Table 4.20 and Table 4.21 shows the detailed metric values obtained from Bogner et al., Runtime Metric Tool.

Bogner et al., Metric Results								
Quality Attribute		**Spring Bookstore Microservices**						**Average**
		api-gateway	**author-service**	**book-service**	**message-service**	**order-service**	**stock-check-service**	
Coupling	AIS	0	1	1	1	1	1	**0.83**
	ADS	3	0	0	0	2	0	**0.83**
	ACS	0	0	0	0	2	0	**0.33**
	MAIDS	0.83						**0.83**
	MACS	1.67						**1.67**
	SIY	0						**0**
Cohesion	SIUC	nan	100%	67%	50%	50%	50%	**63.4%**
	SIDC	nan	0%	67%	67%	100%	100%	**66.8%**
	TSIC	nan	50%	67%	58.50%	75%	75%	**65.1%**
	IUAM	0.67						**0.67**

Table 4.20: Bogner et al., Metric Results for Book Store

Table 4.21: Normalised Bogner et al., Results for Book Store

Normalised Bogner et al., Results for Book Store					
Quality Attribute	**Metric**	**Raw Value**	**Max Value (n=6)**	**Score (0–10)**	**Verdict**
Coupling	AIS (avg)	0.83	5	8.3	Good
	ADS (avg)	0.83	5	8.3	Good
	ACS (avg)	0.33	25	9.8	Good
	MAIDS	0.83	5	8.3	Good
	MACS	1.67	10	8.3	Good
	SIY	0	15	10.0	Good
	Total Coupling Score	—		**8.87**	**Good**
Cohesion	TSIC (avg)	0.65	1	6.5	Acceptable
	IUAM	0.67	1	6.7	Acceptable
	Total Cohesion Score	—		**6.6**	**Acceptable**

Glossary - n : Number of Services, AIS: Absolute Importance of the Service, ADS: Absolute Dependence of the Service, ACS: Absolute Criticality of the Service, MAIDS: Mean Absolute Importance/Dependence in the System, MACS: Mean Absolute Coupling in the System, SIY: Services Interdependence in the System, TSIC: Total Service Interface Cohesion, IUAM: Inverse of Average Number of Used Message

Analysis

MicroValid Metric Suite

The MicroValid results indicate a poor coupling structure, despite the architecture being free of circular dependencies. While the system achieves a perfect SCC TEST score of 10, the DEPENDENCIES COMPOSITION TEST fails with a very low score of 2.2. The details reveal that this is caused by the two services (Order Service and Stock-Check Service) as they both have outward dependencies, while the other three services act purely as providers. This creates a significant structural imbalance that is penalized by the framework.

The system's cohesion is similarly flawed due to severe structural imbalances, even though its interfaces are well-designed. Both the ENTITIES COMPOSITION TEST (score: 2.0) and the RESPONSIBILITIES COMPOSITION TEST (score: 4.6) failed, indicating an uneven distribution of complexity across the services. The Order Service is the largest in terms of data entities, while the Book Service and Author Service handle the most responsibilities.

Normalisation and Scoring Method for Bogner et al., Results

The same steps used to normalise the Bogner et al., results for other studies were used here as well.

The total Bogner et al., Coupling and Cohesion scores for pring Microservices Bookstore are as follows:

- Total Coupling Score: 8.8 (Good)
- Total Cohesion Score: 6.6 (Acceptable)

Bogner et al., Runtime Analysis Tool

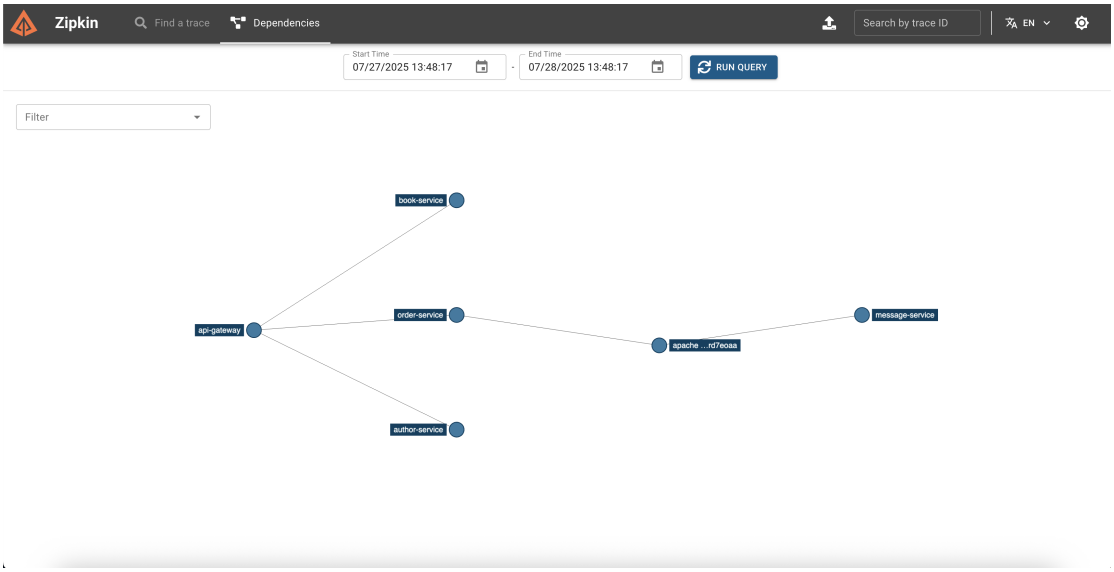


Figure 4.11: Zipkin Dependency graph for Spring Microservices Book Store

The Bogner et al., metrics provide a granular, per-service analysis that offers both positive and critical insights into the system's operational behaviour (Figure 4.11). The system exhibits a highly favorable coupling structure with a perfect SIY score of 0, confirming a lack of interdependencies. However, a significant imbalance is evident in outward dependencies, as the api-gateway has an ADS (Absolute Dependence of the Service) of 3, and the order-service has an ADS of 2, while the other four services have an ADS of 0. This points to api-gateway and order-service as critical, highly dependent components. In contrast to MicroValid, the cohesion metrics are generally positive and consistent. The SIUC (Service Interface Usage Cohesion) scores range from 50% to 100% for all services except the api-gateway. The SIDC (Service Interface Data Cohesion) is also strong, with scores of 67% or higher for all services except the author-service. The combined TSIC (Total Service Interface Cohesion) scores, ranging from 50% to 75%, reflect a good level of overall cohesion.

4.3 Correlation between MicroValid and Bogner Runtime Tool

The correlation coefficients and p-values for coupling and cohesion were calculated by following these steps:

1. **Data preparation:** Extracted total coupling and cohesion scores from MicroValid and Bogner's tool for each of the four applications, forming paired datasets for each metric.

2. **Pearson's correlation coefficient:**

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\,\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

   Computed using `scipy.stats.pearsonr`[2], which also returns the corresponding p-value for testing the null hypothesis $H_0 : r = 0$.

3. **Spearman's rank correlation coefficient:** First, rank each dataset $R(x_i)$ and $R(y_i)$, then apply:

$$\rho = 1 - \frac{6\sum_{i=1}^{n} d_i^2}{n(n^2-1)}, \quad d_i = R(x_i) - R(y_i)$$

   Computed using `scipy.stats.spearmanr`[3], which also provides the p-value for testing $H_0 : \rho = 0$.

4. **p-value interpretation:** The p-value represents the probability of observing a correlation coefficient at least as extreme as the one obtained, assuming the null hypothesis of no correlation is true. A p-value ≤ 0.05 is considered statistically significant.

When comparing MicroValid and Bogner et al., runtime tool, total coupling and cohesion scores for the four evaluated applications (Appendix B), both Pearson's and Spearman's coefficients indicate negative correlations for coupling (Table 4.22) and cohesion (Table 4.23).

[2]https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pearsonr.html
[3]https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html

Table 4.22: Correlation Analysis for Coupling Metrics (MicroValid vs. Bogner et al., Runtime Analysis tool)

Coupling Correlation			
Method	**Coefficient**	**p-value**	**Interpretation**
Pearson's r	$r \approx -0.992$	$p \approx 0.008$	Very strong negative, statistically significant
Spearman's ρ	$\rho = -1.000$	$p < 0.001$	Perfect negative monotonic, statistically significant

Table 4.23: Correlation Analysis for Cohesion Metrics (MicroValid vs. Bogner et al., Runtime Analysis Tool)

Cohesion Correlation			
Method	**Coefficient**	**p-value**	**Interpretation**
Pearson's r	$r \approx 0.505$	$p \approx 0.495$	Moderate positive, not significant
Spearman's ρ	$\rho = 0.400$	$p = 0.600$	Weak positive monotonic, not significant

The coupling attribute exhibits a strong negative correlation, indicating that higher scores in one tool correspond to lower scores in the other. In contrast, the cohesion attribute shows a moderate positive correlation, suggesting that scores from both tools tend to increase in tandem (Figure 4.12).

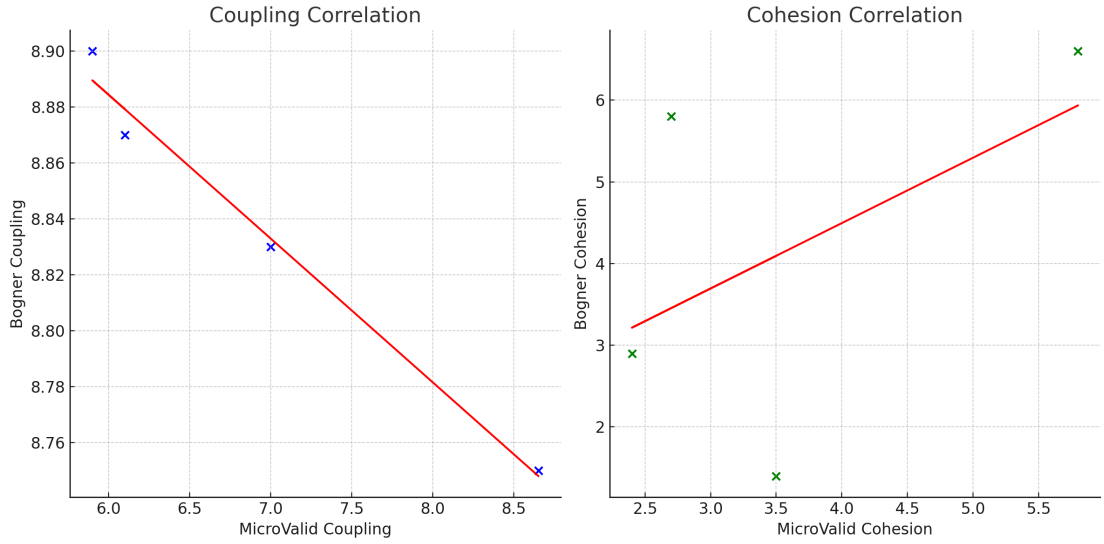


Figure 4.12: Scatter plots with fitted regression lines showing the relationship between MicroValid and Bogner scores for coupling (left) and cohesion (right) across four applications

Inferences

The strong negative correlation found for coupling shows that MicroValid and Bogner et al., runtime metric tool focus on different aspects of service interaction.

MicroValid rates a service as tightly coupled if it has many static dependencies in the source code, even when those dependencies are rarely used at runtime. In contrast, Bogner et al., runtime analysis tool rates coupling based on actual communication during execution, so a service with few static links but heavy runtime calls may appear loosely coupled in MicroValid but highly coupled in Bogner et al., tool. This explains why the two tools give opposite judgments, even when their raw values appear in a similar range.

For cohesion, the relationship between the tools is weaker and less consistent. MicroValid's cohesion measures are based on structural design, such as shared entities or responsibilities in the code. These values reflect how the system was intended to be organized. Bogner et al., cohesion metrics, however, are based on runtime traces and show whether a service's operations are actually used together in practice. The weak correlation suggests that static cohesion does not reliably predict runtime behavior. Therefore, MicroValid's cohesion scores may be best viewed as indicators of design intent, while Bogner et al's results provide a more realistic picture of cohesion during real execution.

4.4 Review of Research Questions

RQ1: Which software quality attributes are predominantly associated with code smells in microservice architectures?

The literature-driven mapping exercise in Chapter 3.5, supported by a formal graph visualization, demonstrated that coupling and cohesion dominate the set of metrics used to identify code smells in microservice systems. Out of 56 unique metrics identified from scholarly and practitioner sources, 25 (45% approx) specifically measured coupling (18 metrics) or cohesion (7 metrics). This evidence not only validates the research focus but also aligns with earlier empirical findings by Bogner, Fritzsch, Wagner and Zimmermann (2019) and Taibi and Lenarduzzi (2018), which identified weak cohesion and excessive coupling as the most recurring forms of architectural design debt in service-based systems.

RQ2: How do MicroValid static coupling and cohesion metrics compare to Bogner et al. runtime metrics when applied to the same microservice systems?

The four case studies (Spring PetClinic, E-Commerce Backend Platform, Ramanujan, and Spring Microservices Bookstore) revealed a very strong and statistically significant negative relationship between MicroValid and Bogner scores, indicating

that static and runtime perspectives frequently produced opposing evaluations of coupling quality, even when both detected similar structural symptoms such as dependency hotspots or cycles. In contrast, cohesion showed a weak-to-moderate positive relationship, though not statistically significant. Here, static and runtime measures occasionally pointed in the same direction, but often disagreed in magnitude: services rated "Failed" by MicroValid sometimes exhibited moderate-to-high runtime cohesion, and vice versa. Taken together, these findings imply that static coupling metrics are not direct predictors of runtime behavior; in fact, their strong negative correlation suggests they are reliable inverse indicators. This means a system assessed as well-coupled by static analysis is likely to be assessed as poorly-coupled by runtime analysis, and vice versa. In contrast, static cohesion scores offer only a limited and inconsistent proxy for runtime cohesion.

4.5 Threats to Validity

Due to time constraints, the evaluation was performed on only four case study systems (Spring PetClinic, E-Commerce Backend Platform, Ramanujan, and the Spring Microservices Bookstore). While these applications vary in domain, technology stack, and size, the small sample set restricts the ability to generalize the findings across the wider spectrum of microservice architectures in practice. Evaluating a larger and more diverse set of systems including large-scale, industry-grade deployments would be necessary to comprehensively validate the alignment or divergence between static and runtime metrics.

The selected case studies are relatively modest in scale compared to complex production environments. Their manageable size made them feasible for detailed static and dynamic analysis but may not fully capture the operational, organisational, and scalability challenges present in much larger systems. As a result, the observed behaviours and metric relationships may not directly translate to large, distributed enterprise applications.

The process of reverse-engineering system models to create the input files for MicroValid involved a number of manual steps, including extracting nanoentities, mapping responsibilities, and defining inter-service relationships. While careful validation was applied, this manual work introduces the possibility of human error. Any inaccuracies in these artefacts could have influenced the static analysis results. Introducing semi-automated extraction techniques or enhanced validation in future work could reduce such risks.

The runtime analysis depended on distributed tracing data collected via Zipkin, which in turn relied on the completeness of the executed test scenarios. If certain API endpoints or workflows were not invoked during the capture period,

the resulting runtime coupling and cohesion metrics may underestimate the actual interactions present in a live environment. Automated and exhaustive workload generation could help ensure more complete runtime traces in future studies.

Finally, differences in the internal definitions and measurement logic of MicroValid and Bogner's runtime metrics could have affected direct comparability. While normalisation was applied to align metric scales, the conceptual constructs measured by each tool are not identical. This inherent methodological difference may contribute to score divergence and should be considered when interpreting the results.

4.6 Summary

This chapter presents an empirical evaluation of static and dynamic metric tools for assessing coupling and cohesion in microservice architectures, using four open-source case study applications: Spring Petclinic, E-Commerce Backend Platform, Ramanujan, and Spring Microservices Bookstore. For each system, the chapter details the deployment setup, prerequisites, and execution of experiments using MicroValid (static analysis) and Bogner's Runtime Analysis Tool (dynamic analysis). The results include per-metric breakdowns, normalisation of runtime measures, and composite coupling/cohesion scores, highlighting points of agreement and divergence between the two approaches.

The chapter also includes a correlation study which reveals that the two tools diverge sharply on coupling assessments, with a very strong negative and statistically significant correlation between their outputs. In contrast, their cohesion evaluations show a weak-to-moderate positive correlation, though not statistically significant, suggesting limited agreement. These results highlight the fundamental methodological differences between static and runtime aspects. MicroValid emphasizes structural balance within the codebase, while Bogner's runtime metrics capture service interactions in execution contexts. Individual case findings identify structural imbalances, dependency hubs, cyclic dependencies, and cohesion inconsistencies, with Bogner's metrics often pinpointing the specific services responsible for quality issues.

Chapter 5

Conclusion and Future Directions

This dissertation set out to address the research gap in evaluating the quality of microservice architectures by directly comparing a static analysis approach, MicroValid, with a runtime analysis approach, proposed by Bogner et al. The study first identified coupling and cohesion as the most relevant architectural quality attributes through a literature-driven mapping of code smells to software metrics that can detect them. Four real-world, open-source microservice systems: Spring PetClinic, E-Commerce Backend Platform, Ramanujan, and the Spring Microservices Bookstore were selected as case studies. For each system, a reverse-engineered JSON architectural model was created to serve as input to MicroValid, and a fully deployed, instrumented version of the system was operated with Zipkin to collect distributed trace data for Bogner's tool.

The results showed sharp divergence between static and runtime tools in detecting coupling issues, with a very strong negative and statistically significant correlation between their outputs. This indicates that while both approaches highlight imbalances and dependency hotspots, they do so in fundamentally different ways, often leading to contrasting evaluations of the same system. By contrast, cohesion assessment revealed weak-to-moderate positive correlations that were not statistically significant. Static cohesion scores frequently proved to be poor proxies for runtime cohesion, with some services appearing well-structured in code analysis but showing fragmented or inconsistent usage patterns during execution.

Beyond these empirical results, this study makes several valuable contributions. It presents the first direct empirical correlation analysis between static coupling and cohesion metrics and their dynamic counterparts on microservice systems. This provides architects and developers with concrete, evidence-based guidance on the strengths and limitations of static metrics. For example, the analysis identifies situations, such as a service with many unused static dependencies or heavy runtime traffic, where static metrics may over or under-estimate actual

coupling or cohesion.

Future research should focus on broadening the scope of this study to include a larger and more diverse set of case studies, enabling validation of the findings across a wider variety of application domains, technology stacks, and system complexities. It should also examine how varying workload patterns and traffic levels influence runtime measurements of cohesion and coupling, enabling quality assessments that reflect actual system usage. Addressing the limitations observed in static analysis, particularly the discrepancies between static and dynamic cohesion measurements, will be crucial. In addition to this, integrating static and runtime metric evaluations into CI/CD pipelines could make architectural quality checks a routine part of the delivery process, providing teams with timely, actionable feedback to help ensure the implemented system remains aligned with its intended design goals. Finally, integrating advanced machine learning techniques to combine static and dynamic data could enable more accurate prediction and validation of optimal microservice boundaries. Advancing these directions will lead to more accurate, scalable, and practical approaches for assessing and improving microservice architectures.

References

Abgaz, Y., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., Jackson, G., Yilmaz, M., Buckley, J. and Clarke, P. (2023), 'Decomposition of monolith applications into microservices architectures: A systematic review', *IEEE Transactions on Software Engineering* **49**(8), 4213–4242.

Al-Debagy, O. and Martinek, P. (2020), 'A metrics framework for evaluating microservices architecture designs', *Journal of Web Engineering* **19**(3–4), 341–370.

Albrecht, A. J. (1979), 'Measuring application development productivity', *Proceedings of the Joint SHARE/GUIDE Symposium* **14**, 83–92. [accessed: 23 Jul 2025].

Arcelli Fontana, F., Avgeriou, P., Pigazzini, I. and Roveda, R. (2019), A study on architectural smells prediction, *in* '2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)', pp. 333–337.

Bigonha, M. A., Ferreira, K., Souza, P., Sousa, B., Januário, M. and Lima, D. (2019), 'The usefulness of software metric thresholds for detection of bad smells and fault prediction', *Information and Software Technology* **115**, 79–92.

Bogner, J., Fritzsch, J., Wagner, S. and Zimmermann, A. (2019), Assuring the evolvability of microservices: Insights into industry practices and challenges, *in* '2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)', pp. 546–556.

Bogner, J., Schlinger, S., Wagner, S. and Zimmermann, A. (2019), A modular approach to calculate service-based maintainability metrics from runtime data of microservices, *in* X. Franch, T. Männistö and S. Martínez-Fernández, eds, 'Product-Focused Software Process Improvement (PROFES 2019)', Vol. 11915 of *Lecture Notes in Computer Science*, Springer, Cham, pp. 567–584.

Bogner, J., Wagner, S. and Zimmermann, A. (2017), Automatically measuring the maintainability of service- and microservice-based systems: a literature review, IWSM Mensura '17, Association for Computing Machinery, New York, NY,

USA, p. 107–115.
URL: *https://doi.org/10.1145/3143434.3143443*

Brito e Abreu, F. and Melo, W. (1996), Evaluating the impact of object-oriented design on software quality, *in* 'Proceedings of the 3rd International Software Metrics Symposium', pp. 90–99.

Chidamber, S. R. and Kemerer, C. F. (1994), 'A metrics suite for object oriented design', *IEEE Transactions on Software Engineering* **20**, 476–493. [online], available: https://doi.org/10.1109/32.295895 [accessed: 23 Jul 2025].

Cojocaru, M., Uta, A. and Oprescu, A.-M. (2019), Microvalid: A validation framework for automatically decomposed microservices, *in* '2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)', pp. 78–86.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L. (2017), Microservices: Yesterday, today, and tomorrow, *in* 'Present and Ulterior Software Engineering', pp. 195–216. [online], available: N/A [accessed: 23 Jul 2025].

Evans, E. (2002), 'Domain driven design: Tackling complexity in the heart of business software'.

Fenton, N. E. and Pfleeger, S. L. (1997), *Software Metrics: A Rigorous and Practical Approach*, 2nd edn, Thomson Publishing.

Filó, T. G., Bigonha, M. A. and Ferreira, K. A. (2024), 'Evaluating thresholds for object-oriented software metrics', *Journal of the Brazilian Computer Society* **30**(1), 315–346.

Fontana, F. A., Mäntylä, M. V., Zanoni, M. and Marino, A. (2016), 'Comparing and experimenting machine learning techniques for code smell detection', *Empirical Software Engineering* **21**, 1143–1191.

Fowler, M. (1999), *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional.

Fowler, M. and Lewis, J. (2014), 'Microservices'. [online], available: https://martinfowler.com/articles/microservices.html [accessed: 23 Jul 2025].

Gupta, R. and Singh, S. K. (2020), Using software metrics to detect temporary field code smell, *in* '2020 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence)', IEEE, pp. 45–49.

Gysel, M., Kölbener, L., Giersche, W. and Zimmermann, O. (2016), Service cutter: A systematic approach to service decomposition, *in* M. Aiello, E. B. Johnsen, S. Dustdar and I. Georgievski, eds, 'Service-Oriented and Cloud Computing', Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 185–200. [online], available: https://doi.org/10.1007/978-3-319-44482-$6_1$2[$accessed : 23Jul2025$].

Halstead, M. H. (1977), *Elements of Software Science (Operating and programming systems series)*, Elsevier Science Inc., USA.

Honglei, T., Wei, S. and Yanan, Z. (2009), The research on software metrics and software complexity metrics, *in* '2009 International Forum on Computer Science-Technology and Applications', pp. 131–136. available: https://doi.org/10.1109/IFCSTA.2009.39 [accessed: 23 Jul 2025].

IEEE Standards Association (1998), 'IEEE Standard 1061 - Standard for a Software Quality Metrics Methodology', `https://standards.ieee.org/ieee/1061/1549/`. [accessed: 23 Jul 2025].

Jerzyk, M. and Madeyski, L. (2023), Code smells: A comprehensive online catalog and taxonomy, *in* N. Kryvinska, M. Greguš and S. Fedushko, eds, 'Developments in Information and Knowledge Management Systems for Business Applications', Vol. 462 of *Studies in Systems, Decision and Control*, Springer, Cham, pp. 543–576.
URL: *https://doi.org/10.1007/978-3-031-25695-$0_2$4*

Kalske, M., Mäkitalo, N. and Mikkonen, T. (2017), Challenges when moving from monolith to microservice architecture, *in* 'International Conference on Web Engineering', Springer, pp. 32–47.

Khomh, F., Di Penta, M. and Guéhéneuc, Y. (2009), An exploratory study of the impact of code smells on software change-proneness, *in* '16th Working Conference on Reverse Engineering (WCRE)', IEEE, pp. 75–84.

Lanza, M. and Marinescu, R. (2006), *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*, Springer Berlin Heidelberg.

Mantyla, M., Vanhanen, J. and Lassenius, C. (2003), A taxonomy and an initial empirical study of bad smells in code, *in* 'International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.', pp. 381–384.

Marinescu, R. (2004), Detection strategies: Metrics-based rules for detecting design flaws, *in* '20th IEEE International Conference on Software Maintenance, 2004. Proceedings.', IEEE, pp. 350–359.

McCabe, T. (1976), 'A complexity measure', *IEEE Transactions on Software Engineering* **SE-2**(4), 308–320.

Moha, N., Guéhéneuc, Y.-G., Duchien, L. and Le Meur, A.-F. (2009), 'Decor: A method for the specification and detection of code and design smells', *IEEE Transactions on Software Engineering* **36**(1), 20–36.

Moha, N., Guéhéneuc, Y.-G., Duchien, L. and Le Meur, A.-F. (2010), 'Decor: A method for the specification and detection of code and design smells', *IEEE Transactions on Software Engineering* **36**(1), 20–36.

Mumtaz, H., Singh, P. and Blincoe, K. (2021), 'A systematic mapping study on architectural smells detection', *Journal of Systems and Software* **173**, 110885.

Newman, S. (2019), *Monolith to microservices: evolutionary patterns to transform your monolith*, O'Reilly Media.

Ntentos, E., Zdun, U., Plakidas, K., Meixner, S. and Geiger, S. (2020), Assessing architecture conformance to coupling-related patterns and practices in microservices, *in* 'European Conference on Software Architecture', Springer, pp. 3–20.

Riel, A. J. (1996), *Object-oriented design heuristics*, Addison-Wesley Longman Publishing Co., Inc.

Saini, N., Kharwar, S. and Agrawal, A. (2014), 'A study of significant software metrics', *International Journal of Engineering Inventions* **3**(12), 1–7. [online], available: N/A [accessed: 23 Jul 2025].

Sarkar, S., Kak, A. C. and Nagaraja, N. (2005), Metrics for analyzing module interactions in large software systems, *in* '12th Asia-Pacific Software Engineering Conference (APSEC'05)', IEEE, pp. 8–pp.

Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S. and Shanbhag, C. (2010), 'Dapper, a large-scale distributed systems tracing infrastructure'.

Singh, G., Singh, D. and Singh, V. (2011), 'A study of software metrics', *IJCEM: International Journal of Computational Engineering & Management* **11**, 22. [online], available: http://www.ijcem.org [accessed: 23 Jul 2025].

Taibi, D. and Lenarduzzi, V. (2018), 'On the definition of microservice bad smells', *IEEE Software* **35**(3), 56–62.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A. and Poshyvanyk, D. (2015), When and why your code starts to smell bad.

Yamashita, A. and Moonen, L. (2013), Exploring the impact of inter-smell relations on software maintainability: an empirical study, *in* 'Proceedings of the 2013 International Conference on Software Engineering', ICSE '13, IEEE Press, p. 682–691.

Zaverdehi, H. Z. (2024), A semi-automated approach for incremental migration from monolithic to microservices architecture, Master's thesis, McMaster University, Hamilton, Ontario, Canada.

Zhong, C., Zhang, H., Li, C., Huang, H. and Feitosa, D. (2023), 'On measuring coupling between microservices', *Journal of Systems and Software* **200**, 111670.

Appendix A

Spring Petclinic MicroValid Input

```
{
  "name": "spring_petclinic_microservices",
  "services": [
    {
      "id": "customers-service",
      "name": "customers-service",
      "nanoentities": [
        "Owner.id",
        "Owner.firstName",
        "Owner.lastName",
        "Owner.address",
        "Owner.city",
        "Owner.telephone",
        "Owner.pets",
        "Pet.id",
        "Pet.name",
        "Pet.birthDate",
        "Pet.type",
        "PetType.id",
        "PetType.name"
      ]
    },
    {
      "id": "vets-service",
      "name": "vets-service",
      "nanoentities": [
        "Vet.id",
        "Vet.firstName",
        "Vet.lastName",
        "Vet.specialties",
        "Specialty.id",
        "Specialty.name"
      ]
```

```
    },
    {
      "id": "visits-service",
      "name": "visits-service",
      "nanoentities": [
        "Visit.id",
        "Visit.date",
        "Visit.description",
        "Visit.petId"
      ]
    },
    {
      "id": "api-gateway",
      "name": "api-gateway",
      "nanoentities": []
    }
  ],
  "relations": [
    {
      "serviceA": "api-gateway",
      "serviceB": "customers-service",
      "sharedEntities": [
        "Owner.id",
        "Owner.firstName",
        "Owner.lastName",
        "Owner.address",
        "Owner.city",
        "Owner.telephone",
        "Pet.id",
        "Pet.name",
        "Pet.birthDate",
        "Pet.type"
      ],
      "direction": "OUTGOING"
    },
    {
      "serviceA": "api-gateway",
      "serviceB": "visits-service",
      "sharedEntities": [
        "Visit.id",
        "Visit.petId",
        "Visit.date",
        "Visit.description"
      ],
      "direction": "OUTGOING"
    }
  ],
```

```
  "useCaseResponsibility": {
    "customers-service": [
      "createOwner",
      "findOwner",
      "findAll",
      "updateOwner",
      "getPetTypes",
      "processCreationForm",
      "processUpdateForm",
      "findPet"
    ],
    "vets-service": [
      "showResourcesVetList"
    ],
    "visits-service": [
      "create",
      "read"
    ],
    "api-gateway": [
      "getOwnerDetails"
    ]
  }
}
```

Appendix B

Total Coupling and Cohesion Scores from MicroValid and Bogner et al., Runtime Analysis Tool

MicroValid		
Case Study	**Coupling**	**Cohesion**
Spring PetClinic	8.65	3.5
E-Commerce Backend Platform	7	2.4
Ramanujan	5.9	2.7
Spring Microservices Bookstore	6.1	5.8

Table B.1: Coupling and Cohesion values for the Microvalid Metric Tool

Bogner et al., Runtime Analysis tool		
Case Study	**Coupling**	**Cohesion**
Spring PetClinic	8.75	1.4
E-Commerce Backend Platform	8.83	2.9
Ramanujan	8.9	5.8
Spring Microservices Bookstore	8.87	6.6

Table B.2: Coupling and Cohesion values for the Bogner et al., Runtime Tool