MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Monoliths Decomposition Techniques for Microservices

Master's Thesis

BC. NORBERT BODNÁR

Brno, Spring 2025

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Monoliths Decomposition Techniques for Microservices

Master's Thesis

BC. NORBERT BODNÁR

Advisor: doc. Bruno Rossi, PhD

Department of Computer Systems and Communications

Brno, Spring 2025


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

During the preparation of this thesis, I used the following AI tools:

- OpenAI ChatGPT for brainstorming ideas, exploratory research, and identifying relevant tools and libraries,
- Anthropic Claude for development assistance, code review and stylistic and structural refinement of the text.

Bc. Norbert Bodnár

Advisor: doc. Bruno Rossi, PhD

Acknowledgements

I would like to thank my advisor, doc. Bruno Rossi, PhD, for his guidance, valuable feedback, and support throughout the development of this thesis. I am also grateful to my family and friends for their encouragement and support during this journey.

Abstract

The transition from monolithic to microservices architecture has become a crucial challenge in modern software development. This thesis addresses the complex process of decomposing monolithic applications into microservices, focusing on the development and evaluation of semi-automatic decomposition techniques. The research presents a comprehensive analysis of existing decomposition approaches, tools, and methodologies, while introducing a novel tool for semi-automatic monolith decomposition. Through extensive case studies and practical implementations, the thesis evaluates different decomposition strategies, identifies common patterns and anti-patterns, and provides guidelines for successful migration. The findings contribute to the understanding of effective decomposition techniques and offer practical insights for organizations undertaking similar architectural transformations. The research combines theoretical analysis with practical implementation, making it valuable for both academic research and industry practitioners.

Keywords

microservices, monolith decomposition, software architecture, system migration, benchmarking, performance evaluation, scalability assessment

Contents

List of Tables

List of Figures

1 Introduction

Organizations initially built large monolithic systems to serve their business needs. As these systems grew, they became difficult to maintain, scale, and deploy [1]. This challenge has driven adoption of microservices architecture [2], which offers better scalability, maintainability, and operational flexibility [3].

The process of decomposing monolithic applications into microservices is full of challenges, trade-offs, and uncertainties [4]. This thesis introduces a tool for supporting the monolith decomposition process, focusing on both static analysis methods for discovering service boundaries and empirical performance benchmarking to validate decomposition quality.

1.1 Motivation

The transition from monolithic to microservices architecture is not straightforward. It requires careful planning, deep understanding of the existing system, and systematic approaches to decomposition. A key challenge is that it is difficult to predict how a proposed decomposition will behave under realistic workload conditions. While microservices offer potential advantages in scalability, maintainability, and operational flexibility, the path toward achieving these benefits is uncertain and often costly [5] [6] [7].

Architectural transitions carry substantial risk. O'Reilly's Technology Trends for 2024 [8] reports that 61% of surveyed enterprises adopted microservices, but 29% encountered difficulties severe enough to return to monolithic architectures. The resulting systems often become more complex to operate, exhibit worse performance, or fail to deliver expected improvements. These outcomes often stem from poorly chosen service boundaries, increased inter-service communication overhead, or unforeseen distributed-system bottlenecks [4]—issues that are difficult to detect through static design analysis alone.

A key source of uncertainty lies in scalability under realistic load. Even when a decomposition looks sound, runtime behavior often diverges once services go into production and handle real traffic. Communication patterns, serialization costs, database contention, and ser-

vice fan-out can degrade performance [1] in ways that are not apparent during design. As a result, architects lack a reliable basis for evaluating the impact of decomposition decisions before committing to a costly migration.

A tool capable of validating target architectures empirically-by generating representative workloads and benchmarking alternative designs-can address this gap. Such a tool enables architects to compare monolithic and decomposed variants using consistent, reproducible metrics and to identify scalability regressions early in the transition process. By grounding architectural decisions in measured behavior rather than assumptions or intuition, organizations can reduce migration risk, prioritize viable decomposition strategies, and make data-driven choices about how to evolve their systems.

1.2 Problem Statement

Despite growing interest in microservices [5] [6] [1], organizations lack systematic ways to evaluate whether proposed decompositions will work before making large-scale architectural changes. Many existing approaches emphasize static analysis techniques for identifying potential service boundaries, yet these techniques provide only limited insight into how a decomposed architecture might behave under realistic runtime conditions. As a result, architects may be required to make decisions without sufficient empirical evidence regarding scalability, performance characteristics, or operational risks.

Furthermore, tools that combine static analysis with automated benchmarking of multiple architectures are scarce [5]. Current practices rely on intuition, ad-hoc experiments, or manual prototyping, making it difficult to compare alternatives fairly or detect performance degradation from decomposition decisions. This lack of structured methodological support may increase the risk of producing architectures that are more complex, less performant, or misaligned with real-world workloads.

1.3 Thesis Objective

The objective of this thesis is to develop a unified, automation-oriented approach that supports both:

- the systematic analysis of monolithic systems to derive meaningful decomposition candidates, and
- the empirical evaluation of these candidates through comparative scalability and performance assessment against the original monolith.

The goal is to provide architects with reproducible, data-driven insights that inform decomposition decisions and reduce uncertainty during architectural evolution. To achieve this, the thesis introduces a tool that integrates static analysis with multi-level scalability assessment [9], enabling structured exploration and validation of alternative architectural designs. While the scalability assessment methodology is adopted from existing research, this work contributes through its automated implementation and integration within a unified decomposition support platform.

1.4 Scope

This thesis presents ServiceSlicer, a research prototype designed to support the exploration and evaluation of microservice decomposition strategies for monolithic systems. The tool focuses on two primary capabilities:

1. **Static-analysis-based decomposition support**, offering guidance on potential service boundaries derived from structural properties of the codebase.
2. **Empirical scalability evaluation**, enabling controlled benchmarking of alternative architectural variants.

ServiceSlicer concentrates on backend service decomposition and performance assessment. It does not address front-end migration, data-migration strategies, or organizational and socio-technical aspects of

microservice adoption. The produced decompositions are exploratory and are intended to inform architectural decision-making rather than serve as production-ready migration outputs. The complete source code accompanies this thesis, and deployment instructions are provided in Appendix B.

Detailed descriptions of these functionalities are presented in later chapters.

1.4.1 Assumptions and Requirements

To keep the scope focused and the evaluation reproducible, the tool operates under several assumptions:

- static analysis is performed on JVM bytecode,
- all system variants can be deployed via Docker Compose,
- architectural alternatives expose a consistent API surface and a public health-check endpoint, and
- benchmarking runs occur in a controlled, single-host environment with representative workload profiles.

These assumptions ensure comparability across experiments while intentionally narrowing the scope to performance and scalability-related aspects of decomposition.

1.5 Thesis Structure

The remainder of this thesis is organized into six chapters that progressively build from foundational concepts to practical demonstration:

Chapter 2 reviews and synthesizes the challenges associated with monolith decomposition and surveys existing methods, including static analysis, dynamic analysis, and domain-driven approaches. It also introduces the M2MDF framework and Multi-Level Scalability Assessment methodology as foundational concepts.

Chapter 3 presents the architecture and design of the ServiceSlicer platform, detailing its components, data flows, and integrated tools for static analysis and benchmarking.

Chapter 4 describes the static code analysis workflow, including dependency extraction, graph-based clustering algorithms, and AI-assisted semantic refinement.

Chapter 5 presents the performance benchmarking workflow, detailing the multi-level scalability assessment methodology and its automated implementation.

Chapter 6 demonstrates the application of ServiceSlicer through experiments on two monolithic applications, assessing its effectiveness in generating decomposition candidates and measuring scalability under load.

Chapter 7 concludes the thesis by summarizing contributions, reflecting on limitations, and outlining directions for future research.

2 Decomposition of Monolithic Systems: Challenges and Methods

Decomposition is a central but difficult step in migrating from monolithic to microservice architectures. This chapter examines why decomposition is challenging, introduces a framework that organizes existing research, surveys the principal analysis techniques, and identifies gaps that motivate the design of ServiceSlicer. We begin by exploring the multifaceted nature of decomposition challenges, then introduce the M2MDF framework as a lens for understanding the decomposition lifecycle, before examining specific analysis methods and the scalability assessment methodology that enables empirical validation.

2.1 The Decomposition Challenge

Service decomposition determines how a monolith should be broken apart and what services should exist in the target system. The quality of these boundary decisions has a direct impact on the feasibility, cost, and long-term success of the migration [5]. Poorly defined services can lead to data fragmentation issues, excessive inter-service communication, operational complexity, and degraded runtime performance. Decomposition is not only a technical task but a socio-technical and operational one, with three classes of challenges generally arising.

2.1.1 Technical Challenges in Finding Boundaries

Identifying meaningful service boundaries is inherently difficult because monoliths evolve over long periods and accumulate dense webs of dependencies [10] [11]. This leads to:

- **Tightly coupled modules** where isolating functionality often breaks hidden assumptions or introduces circular dependencies.
- **Erosion of architectural structure**, making it unclear where natural boundaries should lie.

- **Granularity decisions**, as services that are too coarse-grained preserve monolithic bottlenecks, while overly fine-grained services create excessive communication overhead.

- **Shared data models**, where a single database enforces consistency across components that were never designed to function independently. Splitting such schemas requires determining data ownership, exploring patterns like sagas [12], and accepting weaker consistency models.

These challenges mean that a decomposition cannot be derived solely from conceptual design ideals—it must account for the system's actual structural and behavioral constraints. These technical difficulties are compounded by operational and organizational constraints that extend beyond code structure.

2.1.2 Operational and Organizational Challenges

Moving from a monolithic system to a distributed architecture adds significant operational complexity. Microservices rely on practices and tools that differ greatly from those used in monolithic environments. Operating many independently deployed services requires strong monitoring, distributed tracing, log aggregation, automated deployment pipelines, service discovery, and resilience mechanisms such as circuit breakers [13]. Debugging becomes harder because failures often span multiple services, networks, or data stores. Without sufficient operational maturity, these demands can reduce reliability, increase recovery times, and raise the cognitive load on teams.

These operational demands are closely tied to organizational factors. Conway's Law [14] states that a system's architecture tends to mirror the structure of the teams that create it. This means that introducing a distributed microservice architecture without adjusting team boundaries, communication patterns, or ownership roles can lead to confusion and friction. Even if a decomposition looks technically sound, it may still fail in practice if it requires teams to work across unclear interfaces or undermines existing responsibilities. For a decomposition to be successful, the organization must ensure that team structures and communication patterns align with the intended

service boundaries [15]. Even when technical and organizational challenges are addressed, practitioners face a critical tooling gap.

2.1.3 Tooling and Evaluation Gaps

Although various tools exist for static code analysis and dependency visualization, end-to-end support for the full decomposition process is still limited. Most tools focus on identifying potential service boundaries based on structural indicators such as coupling, cohesion, or community detection results [16] [11]. However, these techniques rarely go beyond boundary discovery and offer little help in determining whether the proposed services are practical when considering runtime behavior, performance characteristics, or operational constraints. A recent systematic review [5] also notes the lack of validated and widely accepted metrics or benchmarks for assessing microservice quality, making it difficult to compare different extraction methods in a consistent and rigorous way.

The review further observes that very little research examines how extracted services behave once implemented and executed in real or simulated environments. Empirical evaluation after deployment—where the actual suitability of a microservice design becomes apparent under realistic workloads—remains largely unexplored. Yet this is the point at which the strengths and weaknesses of a proposed decomposition can be most clearly understood. This empirical validation gap is a central motivation for this thesis.

Given these multifaceted challenges, a structured approach to decomposition is essential. The following section introduces a framework that consolidates existing research into a coherent process model, providing the conceptual foundation for understanding where current approaches succeed and where critical gaps remain.

2.2 The M2MDF Framework: A Structured Decomposition Process

The Monolith to Microservices Decomposition Framework (M2MDF) [5] addresses the fragmentation of existing approaches by providing a consolidated, end-to-end process model derived from a systematic re-

view of 35 studies. Rather than proposing yet another decomposition algorithm, M2MDF organizes the diverse techniques found across the literature into six coherent phases spanning the entire decomposition lifecycle.

2.2.1 Framework Phases

Phase I — Input Collection This phase gathers artefacts that describe the monolith, including design models, source code, execution logs, and version histories. These heterogeneous inputs provide complementary views of the system, yet the review finds that no existing approach combines all input types in a single method.

Phase II — Monolith Analysis Collected inputs are analyzed using four principal methods: domain analysis, static analysis, dynamic analysis, and version analysis. These techniques uncover structural and behavioral characteristics such as coupling patterns, runtime interactions, and system evolution, forming the analytical basis for identifying service boundaries.

Phase III — Microservices Identification Candidate services are derived either through rule-based approaches, where human experts define partitioning criteria, or through clustering methods that group system components using machine-learning or graph-based algorithms. This phase produces initial boundary proposals that reflect the system's technical or semantic structure.

Phase IV — Microservices Optimisation Initial candidates are refined using techniques such as genetic algorithms or neural-network-based clustering to improve cohesion, reduce coupling, or search for more optimal service boundary configurations. Not all studies implement this phase, highlighting a gap in automated refinement support.

Phase V — Microservices Evaluation Evaluation assesses the quality of proposed services using case studies, examples, controlled experiments, or metric-based comparisons. Commonly used metrics include

cohesion, coupling, precision, and recall, as well as non-functional indicators. This phase provides empirical validation for decomposition decisions, yet standardized benchmarks remain limited.

Phase VI — Microservices Deployment The final phase examines how extracted microservices behave when deployed in real or emulated environments. The review notes that this phase is significantly underrepresented in current research, with only one study attempting deployment-based validation, despite its critical role in revealing runtime suitability and scalability issues.

M2MDF's synthesis reveals a critical gap: while Phases I–V (input collection through evaluation) are well-represented in the literature, **Phase VI (deployment-based validation) is nearly absent**. Only one of the 35 reviewed studies attempted to evaluate extracted microservices in a deployed environment [5]. This gap is particularly problematic because static metrics and theoretical analysis cannot predict how a decomposition will behave under realistic operational conditions. Addressing this gap requires both a methodological foundation for empirical evaluation and automated tooling to make such evaluation practical.

2.3 Existing Solution Approaches

Having established M2MDF as our organizing framework, we now examine the specific techniques that populate its Phase II (Monolith Analysis). These methods differ in the types of information they analyze—domain models, code structure, runtime behavior, or version history—and in the granularity of boundaries they produce. Understanding their strengths and limitations provides context for the hybrid approach adopted in ServiceSlicer.

2.3.1 Domain-Driven Analysis

Domain analysis approaches focus on understanding a monolithic system through the models and artefacts produced during its original requirements and design phases. Instead of examining the code directly, these methods rely on higher-level representations of system

structure and behavior, such as data flow diagrams, activity diagrams, use case descriptions, entity relationship diagrams, and other UML artefacts.

The central goal of domain analysis is to identify coherent functional areas—domains of interest—that naturally reflect how the system operates and how its capabilities are organized. These domains capture key business concepts, workflows, and responsibilities, making them strong candidates for microservice boundaries. Because of this, domain analysis is frequently referred to as model-driven or domain-driven decomposition.

The most widely used method in this category is Domain-Driven Design (DDD [17]), which introduces concepts such as bounded contexts, aggregates, and domain models to align system structure with business capabilities. DDD produces boundaries that are meaningful and stable but may require substantial domain expertise—something often missing in legacy systems.

A more recent method is Actor-Driven Decomposition (ADD [18]), which shifts the focus from business concepts to the actors interacting with the system. By analyzing user goals, workflows, and operational scenarios, ADD identifies behavioral patterns that reveal how the system is actually used. This makes ADD a useful complement to DDD, grounding decomposition decisions in real operational contexts. While domain-driven approaches produce semantically meaningful boundaries, they require substantial domain expertise and may not reflect the system's actual structural constraints.

2.3.2 Static Code Analysis

Static analysis approaches use structural information extracted directly from the codebase. They typically analyse package structures, class dependencies, call graphs, data access patterns, or architectural layers. Systems are often represented as graphs, and clustering or community-detection algorithms are applied to group elements that are tightly coupled and may form microservice candidates.

These methods have several benefits: they are deterministic, easy to repeat, and work even when runtime data or domain documentation is missing. They are especially useful for legacy systems where observing real execution behavior is difficult. However, static analysis

provides only a structural view of the system. It may group parts of the code that are structurally connected but not meaningful from a business perspective, or miss important runtime interactions that only appear under actual load. Static analysis excels at processing legacy systems where documentation is missing, but it captures only structural relationships—not business semantics or runtime behavior.

2.3.3 Dynamic Runtime Analysis

Dynamic analysis techniques observe a system's behavior at runtime by collecting execution traces, service invocation sequences, or user interaction logs. These methods group components based on how they are used together during typical workloads, identifying behavioral coupling that may not appear in static dependency graphs.

Dynamic approaches capture real execution paths and operational hotspots, making them effective for detecting runtime clusters and usage patterns. They are particularly useful for large systems where static dependencies obscure actual behavior. However, they depend on high-quality trace data; if the collected workload does not reflect the full range of system behavior, the resulting decomposition may be incomplete or biased. Instrumentation overhead, limited observability, and the need for representative execution environments also restrict their applicability in some industrial settings. Dynamic approaches reveal operational patterns invisible in static code, but they require comprehensive trace data that captures the full range of system behavior—something that may be difficult to obtain in practice.

2.3.4 Version History Analysis

Version analysis approaches study the evolution of a monolithic system by examining how its codebase changes over time. Instead of focusing only on the system's current structure or behavior, these methods use historical information—typically from version control systems—to identify components that tend to evolve together. The underlying assumption is that artifacts that frequently change together often implement related functionality and may therefore belong to the same microservice.

Version analysis is rarely used on its own, but it is often combined with static analysis to enrich structural dependency graphs with evolutionary signals [5]. For example, a call graph produced through static analysis can be enriched with evolutionary coupling data, such as files or classes that were co-committed during development. This helps uncover relationships that may not be obvious from the code structure alone but become clearer when examining long-term change patterns.

By looking at how the system has changed over time, version analysis adds a useful historical perspective to static and dynamic techniques. It can reveal groups of components that naturally belong together because they have evolved in similar ways, offering insights that may not appear from code structure or runtime behavior alone. Version analysis provides a temporal dimension to decomposition but is rarely sufficient on its own, typically serving as an enrichment signal for static or dynamic methods.

Each of these analysis techniques addresses specific aspects of boundary identification. However, as the M2MDF review revealed, the critical weakness in the field lies not in the analysis methods themselves but in the near-complete absence of deployment-based validation. Even well-identified service boundaries remain unverified until the resulting architecture is tested under realistic conditions.

2.4 Evaluating Decomposition Quality Through Scalability Assessment

Addressing this validation gap requires a systematic methodology for comparing architectural variants under realistic load. Multi-Level Scalability Assessment [9] provides such a methodology, enabling engineers to compare alternative architectures and understand how they scale at different levels of granularity. This section introduces the key concepts from this methodology that ServiceSlicer implements; subsequent chapters reference these definitions.

2.4.1 Multi-Level Scalability

Rather than treating scalability as a single aggregate property, the methodology analyzes system behavior at multiple levels of granularity: **system-level** metrics capture overall throughput and response time across the entire application, while **operation-level** metrics examine individual API endpoint behavior. This multi-level perspective enables engineers to identify not only whether a system scales, but which specific operations limit scalability.

2.4.2 Baseline Threshold Definition

The methodology establishes performance expectations through a **baseline threshold** for each operation. This threshold is computed from the baseline architecture (typically the original monolith) running at the minimum load level:

$$\Gamma_j^0 = \mu_j^0 + 3\sigma_j^0 \tag{2.1}$$

where μ_j^0 is the mean response time and σ_j^0 is the standard deviation for operation j at baseline load. This threshold serves as the performance reference against which all architectures—including the baseline itself at higher loads—are evaluated. An operation passes its scalability check if its mean response time remains below this threshold.

2.4.3 Domain Metric

The **Domain Metric (DM)** quantifies overall scalability satisfaction as a single value between 0 and 1. It aggregates operation-level results across all load levels, weighted by both the probability of each load level occurring and the relative frequency of each operation in the workload. A DM of 1.0 indicates perfect scalability (all operations meet their thresholds at all loads), while lower values indicate degradation. This metric enables direct comparison between architectural variants: a higher DM indicates better overall scalability for the given operational profile.

2.4.4 Supporting Metrics

The methodology defines additional metrics that provide diagnostic insight:

- **Scalability footprint**: The maximum load level at which an operation consistently meets its threshold, indicating the operation's capacity limit.
- **Scalability gap**: The cumulative weighted contribution of threshold violations across load levels, quantifying how much load contributes to failure.
- **Performance offset**: The severity of threshold violations, measuring how far response times exceed acceptable limits.

2.4.5 Operational Profile

The methodology operates by defining an **operational profile** that captures realistic usage patterns through three components: behavior models (sequences of API operations representing user workflows), behavior mix (relative frequencies of different workflows), and load distribution (discrete concurrency levels and their probabilities of occurrence). This probabilistic characterization ensures that scalability metrics emphasize commonly encountered scenarios rather than treating all load levels uniformly.

Applying this methodology manually is labor-intensive, requiring deployment automation, workload generation, metrics collection, and multi-level analysis. Integrating this methodology into an automated toolchain is a key contribution of this thesis.

2.5 Summary and Positioning of ServiceSlicer

This chapter outlined the multifaceted challenges of monolith decomposition and surveyed existing solution approaches. Technical obstacles—tightly coupled code, unclear boundaries, shared data models—make it difficult to derive clean service partitions, while operational and organizational factors introduce additional risks when transitioning to distributed architectures.

Existing decomposition methods offer complementary perspectives: static analysis captures structural dependencies, dynamic analysis reveals runtime behavior, domain-driven methods identify semantically coherent boundaries, and version analysis uncovers evolutionary coupling. Each contributes valuable insights, yet none addresses the end-to-end decomposition problem in isolation.

The M2MDF framework synthesizes these techniques into a six-phase process model, but highlights a critical gap: **deployment-based validation is nearly absent from current research**. While static metrics and clustering algorithms can suggest candidate service boundaries, they cannot predict how those boundaries will behave under realistic operational conditions. Multi-Level Scalability Assessment provides a methodological foundation for empirical evaluation, but its manual application is impractical for iterative decomposition exploration.

ServiceSlicer addresses these gaps by integrating static analysis, semantic refinement, and automated scalability assessment into a unified platform. Rather than proposing new clustering algorithms or evaluation metrics, this thesis contributes an **automated implementation** of existing methodologies, making systematic, data-driven decomposition accessible to practitioners. The following chapter presents ServiceSlicer's architecture and explains how it operationalizes the M2MDF framework with particular emphasis on Phase VI—deployment-based validation through automated benchmarking.

3 ServiceSlicer — System Architecture & Design

This chapter operationalizes the concepts introduced in Chapter 2 by mapping the M2MDF framework phases and the Multi-Level Scalability Assessment methodology to concrete system components and workflows. The scalability metrics defined in Chapter 2—DM, baseline thresholds, scalability footprint, and operational profiles—are implemented through the platform's benchmarking infrastructure, while the decomposition approaches (static analysis, domain-driven design, actor-driven decomposition) are realized through the static analysis workflow.

The following sections describe ServiceSlicer's architecture, key components, and data flows, explaining how the platform integrates static analysis and empirical benchmarking to support monolith decomposition.

3.1 Overview

ServiceSlicer is a research tool designed to assist software engineers in decomposing monolithic Java applications into microservice architectures. Rather than relying solely on static code inspection or expert intuition, the platform integrates **static dependency analysis** with **empirical performance testing** to generate quantitative, evidence-based insights into how different service boundaries affect system behavior.

By combining these two perspectives, ServiceSlicer directly addresses a central challenge in software architecture: assessing whether a proposed decomposition will deliver measurable improvements in performance and scalability compared to the original monolith. This enables architects to move beyond speculative design and validate decomposition decisions using reproducible data collected under realistic workloads.

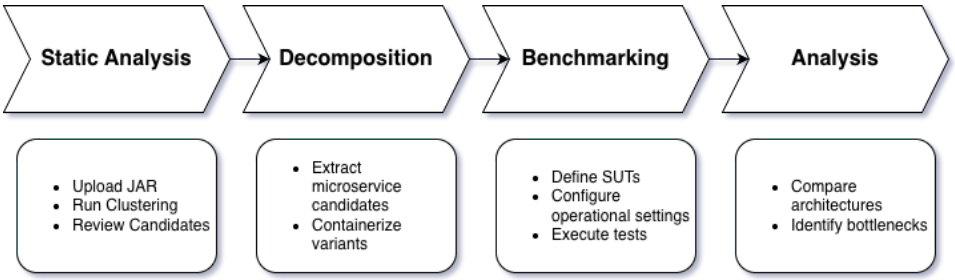

Figure 3.1: High-level end-to-end workflow of the ServiceSlicer platform

3.2 Workflows

ServiceSlicer orchestrates two primary workflows that together support the end-to-end decomposition and validation process.

The **static analysis workflow** accepts a monolithic application artefact and produces multiple decomposition candidates through a combination of structural analysis and semantic refinement. The workflow operates asynchronously: dependency extraction produces a graph representation, clustering algorithms identify candidate service boundaries, and optional AI-based refinement applies domain-driven or actor-driven decomposition strategies. Throughout this process, intermediate and final results are persisted to enable traceability and incremental exploration. Users retrieve consolidated decomposition results through queries that aggregate structural metrics, cluster assignments, and semantic annotations.

The **benchmarking workflow** evaluates the runtime behavior of deployable system architectures under realistic load. Users supply deployment configurations and operational profiles that define expected usage patterns. The workflow operates in three phases: validation confirms conformance to the operational profile, execution deploys systems and runs controlled load tests at multiple concurrency levels, and analysis compares performance metrics against baseline thresholds to compute scalability indicators. Performance data collected during testing is aggregated and persisted, enabling comparative analysis across architectural variants through visualization and metric queries.

Both workflows follow an event-driven model in which user-initiated commands trigger asynchronous processing pipelines. State transitions and intermediate outputs are persisted at each stage, ensuring traceability and enabling recovery from failures. While the workflows are independent—one produces candidate decompositions, the other validates them empirically—they are complementary: architects use static analysis to identify boundaries and benchmarking to assess their viability under realistic conditions. Detailed descriptions of each workflow, including the sequence of operations and interactions between components, are provided in Chapters 4 and 5.

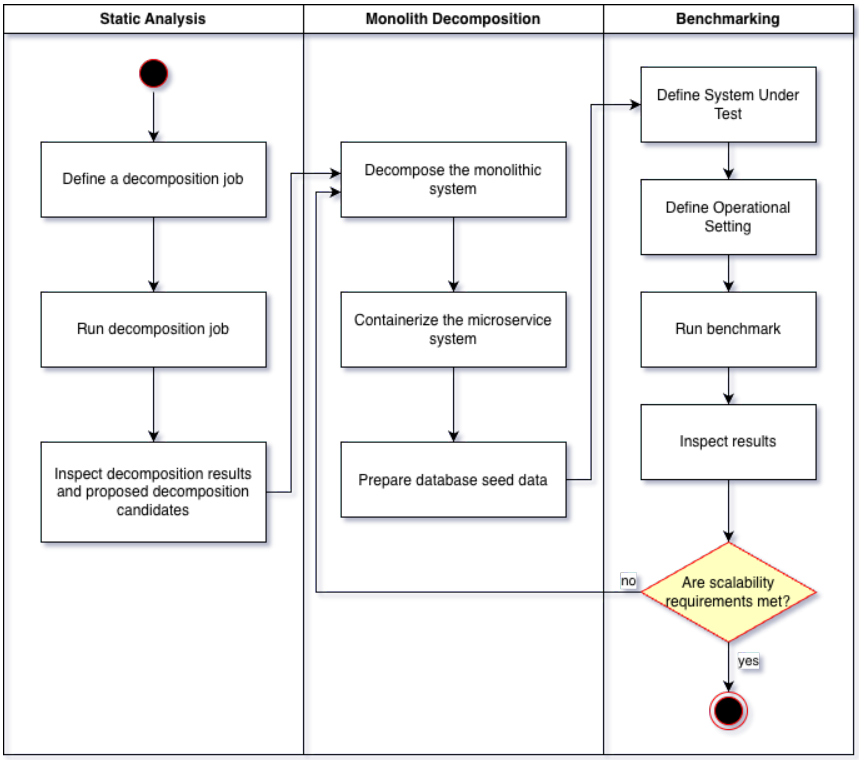

Figure 3.2: Activity diagram showing static analysis and benchmarking workflow stages

3.2.1 Alignment with the M2MDF Framework

ServiceSlicer addresses several critical gaps identified in the M2MDF review by providing integrated tooling support across multiple framework phases. As illustrated in Figure 3.3, the platform implements functionality spanning Phases I through VI, with particular emphasis on the underrepresented deployment-based evaluation phase.

The **static analysis workflow** implements M2MDF Phases I–IV. In **Phase I** (**Input Collection**), ServiceSlicer accepts compiled JAR artefacts, focusing on bytecode-level analysis rather than requiring source code or design documentation. For **Phase II** (**Monolith Analysis**), the platform employs static analysis through bytecode inspection to extract class-level dependencies and construct a complete structural graph. **Phase III** (**Microservices Identification**) leverages graph-based clustering algorithms to identify candidate service boundaries, supplemented by AI-assisted semantic analysis to generate domain-driven and actor-driven decomposition candidates. The platform provides limited support for **Phase IV** (**Optimisation**) through AI-assisted refinement suggesting cluster merges, splits, and naming improvements.

The **benchmarking workflow** implements M2MDF Phases V–VI. For **Phase V** (**Evaluation**), ServiceSlicer computes structural quality metrics including service coupling and size distribution, enabling quantitative comparison of decomposition candidates. Most notably, ServiceSlicer directly addresses the Phase VI gap identified in the M2MDF review: **deployment-based validation**. The platform automates the complete benchmarking lifecycle—deploying architectural variants via container orchestration, executing realistic workload profiles, collecting performance metrics, and computing multi-level scalability indicators. This empirical evaluation capability enables architects to validate decomposition decisions using measured system behavior under realistic conditions, moving beyond the speculative approaches that characterize existing research.

Having established ServiceSlicer's positioning within the M2MDF framework, the following sections detail the system's operational constraints, architectural structure, and component design that enable this end-to-end decomposition support.

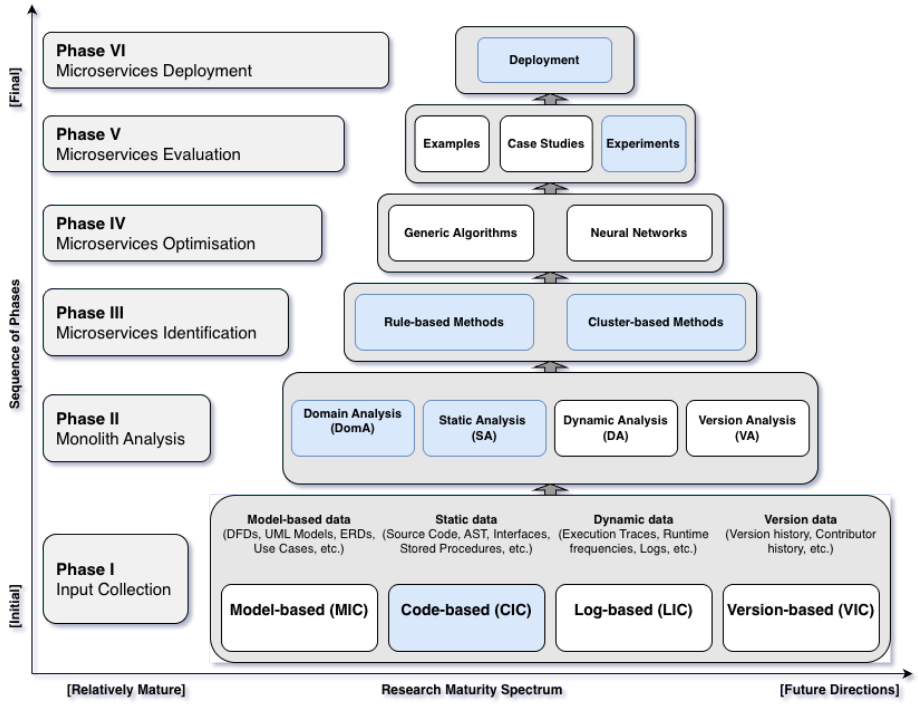

Figure 3.3: The M2MDF Framework Phases (reproduced from [5]) with highlighted phases implemented by ServiceSlicer

3.3 System Assumptions and Operational Constraints

The design of ServiceSlicer is grounded in several assumptions and constraints that define the boundaries within which the platform operates. These conditions ensure that analyses and benchmarking results are both feasible and reproducible. Before detailing the system's internal mechanisms, it is therefore necessary to make explicit the expectations placed on user-provided artefacts, execution environments, and supported technologies.

First, the static analysis workflow assumes that the application under study is a JVM-based system. Dependency extraction relies on bytecode inspection and thus requires the monolith to be provided as a compiled `.jar` file. Source-level parsing or non-JVM binaries fall outside the supported scope.

Second, all architectural variants evaluated during benchmarking must be deployable through Docker Compose. This requirement provides a uniform execution model that allows ServiceSlicer to control startup, resource allocation, health checks, and teardown in a consistent and automated manner. Each System Under Test (SUT) must also expose a REST API described by an OpenAPI specification. This specification is used to construct workload models and to derive the operational setting needed for multi-level scalability assessment. For data initialization, the platform currently supports only PostgreSQL as the database engine for loading seed datasets during benchmark execution.

Operational settings are expected to reflect realistic system usage. Users should derive actor behavior, workload distributions, and concurrency profiles from empirical observations—such as logs or monitoring data—whenever possible.

Finally, benchmarking must be performed in a controlled single-host environment to ensure fair comparability across SUTs. Running all architectures under identical hardware and resource conditions eliminates environmental noise and isolates the architectural effects under study.

These assumptions and constraints define the envelope within which ServiceSlicer produces consistent, interpretable, and reproducible results. Subsequent sections build upon this foundation to describe the system's internal architecture, workflows, and analytical capabilities.

3.4 High-Level Architecture

ServiceSlicer is implemented as a web-based application that orchestrates the entire decomposition and evaluation workflow. It consists of multiple interconnected components responsible for dependency extraction, graph processing, automated deployment of architectural variants, and multi-level scalability assessment. Users interact with the platform through a browser interface, while backend services handle analysis, benchmarking execution, and results aggregation. All generated artefacts—such as dependency graphs, metrics, benchmark outputs, and operational profiles—are stored persistently, ensuring

traceability, repeatability, and consistent comparison across multiple decomposition candidates.

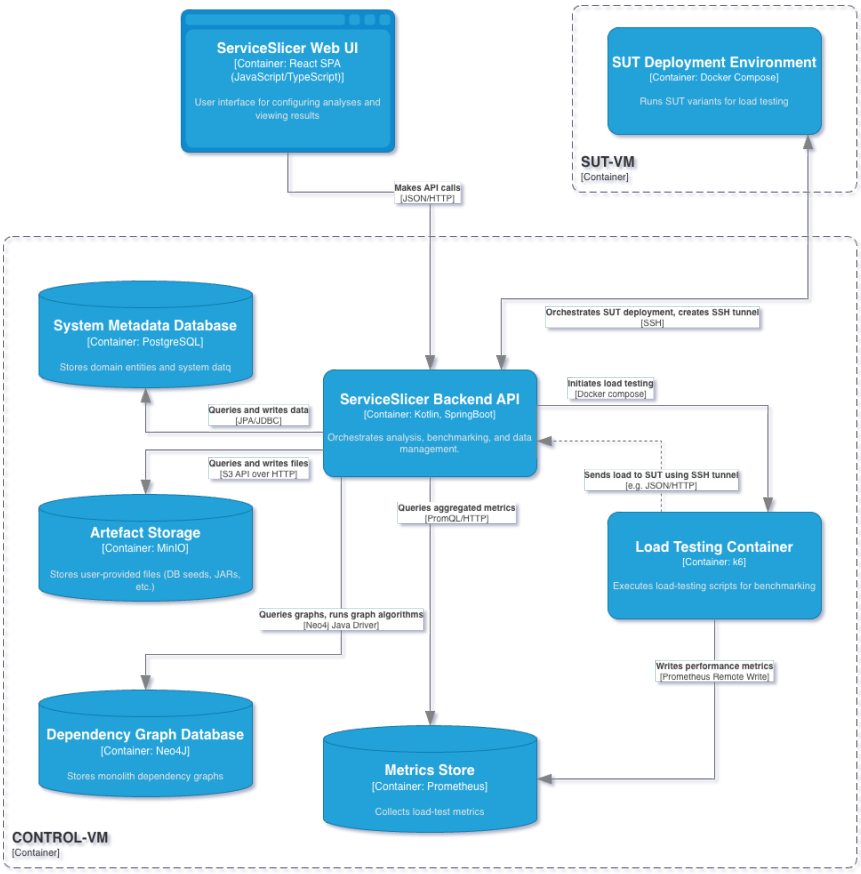

Figure 3.4: High-Level Architecture of ServiceSlicer

ServiceSlicer integrates several cooperating components and external tools, as depicted in Figure 3.4. The system follows a client-server architecture: a React-based frontend communicates with a Kotlin/Spring Boot backend via a REST API. The backend orchestrates all analytical and benchmarking workflows, coordinating interactions with four persistent storage systems—PostgreSQL for domain entities and workflow state, Neo4j for dependency graphs, MinIO[1] for large binary artefacts, and Prometheus[2] for performance metrics. During bench-

1. `https://min.io/`
2. `https://prometheus.io/`

marking, the backend automatically deploys SUTs using Docker Compose, executes load tests with k6[3], and collects metrics for comparative analysis. The architecture is designed to support both local execution (for development) and remote deployment (for reproducible testing environments).

3.5 Component Overview

ServiceSlicer's architecture comprises three distinct layers, each serving a specific role in the decomposition and evaluation workflow. This section describes the responsibilities of components within each layer and their interactions.

3.5.1 Presentation and Application Layer

The presentation and application layer implements the core business logic and user interface for decomposition workflows.

Frontend The frontend is a React-based single-page application that provides the user interface for configuring analyses, uploading artefacts, running benchmarks, and inspecting results. It communicates with the backend via a type-safe, auto-generated API client and renders interactive views such as dependency graphs and performance charts.

Backend The backend is a Kotlin/Spring Boot service that orchestrates all workflows in ServiceSlicer. Using a modular, hexagonal architecture (Figure 3.5), it coordinates interactions with databases, storage systems, and external tools. It implements the domain logic for decomposition jobs, benchmark execution, operational profile management, and scalability analysis, serving as the central engine that executes static analysis, manages load-test automation, aggregates metrics, and exposes the REST API consumed by the frontend.

3. `https://grafana.com/docs/k6/latest/`

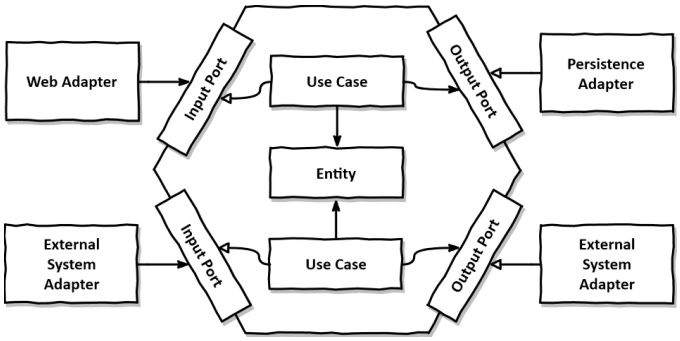

Figure 3.5: Hexagonal Architecture, reproduced from [19]

3.5.2 Data Storage Layer

The data storage layer provides persistent storage for domain entities, dependency graphs, artefacts, and performance metrics.

PostgreSQL — Relational Database PostgreSQL persists all core domain entities, including decomposition jobs, benchmark configurations, operational settings, and aggregated performance results. It provides transactional consistency and structured querying, while supporting JSON fields for flexible data structures. It serves as the authoritative store for workflow state and as the source of truth for all user-visible results.

Neo4j — Graph Database Neo4j stores the dependency graph extracted from the monolithic application, representing classes and their relationships. It executes community detection algorithms to identify clusters that may represent candidate microservices. Graph-native storage and analytics enable efficient traversal, pattern querying, and algorithmic decomposition—capabilities that would be cumbersome in a relational model.

MinIO — Object Storage MinIO serves as an S3-compatible object storage system for large artefacts such as JAR files, OpenAPI specifications, Docker Compose configurations, and seed data. Files are uploaded and downloaded via presigned URLs to avoid routing large

payloads through the backend, efficiently storing and retrieving artefacts that are too large or too unstructured for relational storage.

Prometheus — Time-Series Metrics Store Prometheus collects fine-grained time-series metrics produced by K6 during load testing, such as response times, percentiles, and error rates. It enables measurement of runtime behavior under controlled load and supports computation of multi-level scalability indicators.

3.5.3 Analysis and Testing Infrastructure

The analysis and testing infrastructure layer provides the execution environment for load testing and deployment automation.

Docker Compose — Deployment Orchestration Docker Compose deploys each System Under Test in a reproducible, isolated environment. It ensures that all candidate architectures—whether monolithic or microservice-based—can be started, validated, and torn down automatically, guaranteeing consistent testing conditions and enabling automated end-to-end benchmark execution.

K6 — Load Testing Engine K6 executes behavior-model-driven test scenarios during benchmarking, generating realistic workloads that simulate user interactions according to defined operational profiles. Together with Prometheus, it measures the runtime behavior of each architectural candidate under controlled load.

This layered architecture separates concerns between user interaction, persistent state management, and execution infrastructure, enabling independent evolution of each layer while maintaining clear contracts between them.

4 Static Code Analysis Workflow

This chapter details the implementation of M2MDF Phases I–III (Input Collection, Monolith Analysis, and Microservices Identification) introduced in Chapter 2. It describes how ServiceSlicer extracts structural dependencies from Java bytecode, constructs dependency graphs, and applies clustering algorithms to identify candidate service boundaries. The semantic refinement step operationalizes the domain-driven and actor-driven decomposition approaches discussed in Section 2.2.

The static code analysis workflow provides architects with data-driven insights into how the system's components interact, enabling informed decisions about service partitioning.

4.1 Workflow Overview

The workflow is structured around the concept of a decomposition job, which encapsulates all metadata, configuration parameters, and intermediate results associated with a single analysis run. Each job progresses through a clearly defined sequence of processing stages, with all outputs persisted to ensure traceability, auditability, and comparability across multiple decomposition attempts. The workflow consists of four sequential phases: job initialization, dependency extraction, algorithmic clustering, and semantic refinement.

4.1.1 Phase 1: Job Initialization and Configuration

Users initiate a decomposition job through the web interface (see Appendix C, Figure C.1) by uploading a compiled JAR file representing the monolithic application and specifying configuration parameters such as the base package namespace and optional exclusion patterns. The ability to exclude specific packages or patterns addresses a practical concern: many codebases contain automatically generated artefacts—such as persistence layer code, API schema definitions, or database access objects—that do not reflect meaningful domain relationships. Including such artefacts in the dependency analysis would introduce structural noise and potentially distort the resulting service boundary recommendations. Once submitted, the job is queued for

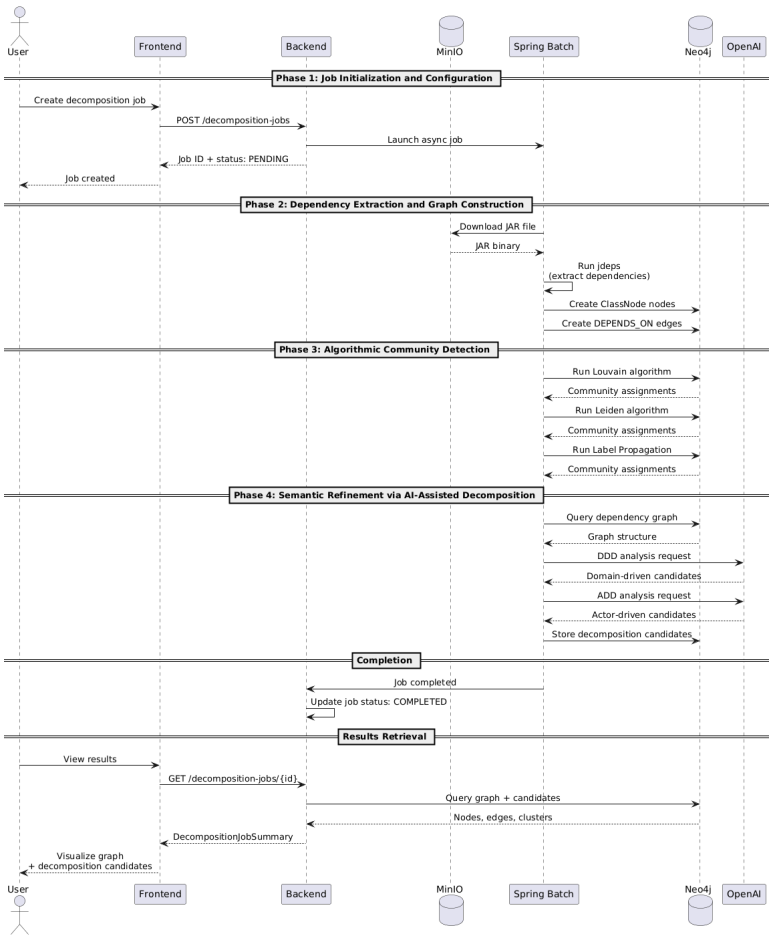

Figure 4.1: Static Analysis Sequence Diagram

asynchronous processing, enabling the analysis of large codebases without blocking the user interface.

4.1.2 Phase 2: Dependency Extraction and Graph Construction

The dependency extraction phase analyzes the bytecode of the uploaded application to identify class-level structural relationships. Rather than performing source-level parsing—which would require access to source code and introduce complications related to language version compatibility—the implementation operates on compiled bytecode using the JDK's dependency analysis utilities. This approach extracts

dependencies at class granularity and produces a graph representation in a standard interchange format.

The extraction process applies the user-specified inclusion and exclusion patterns to filter the analysis scope. Anonymous inner classes and compiler-generated synthetic constructs are normalized to their enclosing classes to avoid graph fragmentation. The resulting dependency information is structured as a directed graph $G = (V, E)$, where vertices V represent classes and directed edges E represent dependencies such as method invocations, field references, and type relationships.

This graph is persisted in a graph database, enabling efficient execution of subsequent graph-theoretic analyses. The database representation uses nodes to model individual classes, with each node storing the fully qualified class name and a reference to the parent decomposition job. Directed edges capture the "depends-on" relationship between classes. The transactional persistence strategy ensures that each decomposition job maintains an isolated graph instance, preventing interference between concurrent or historical analyses.

4.1.3 Phase 3: Algorithmic Community Detection

Following dependency extraction, the workflow applies graph-based community detection techniques to identify structurally cohesive clusters of classes that may represent candidate microservice boundaries. Community detection algorithms partition a graph into densely connected subgraphs while minimizing edges between partitions—a structural property that aligns with the architectural principle of high cohesion and low coupling.

The system applies three distinct community detection algorithms to the dependency graph, each offering different trade-offs between computational efficiency, partition quality, and sensitivity to graph topology:

1. **Leiden Algorithm** [20]: A hierarchical community detection method that optimizes modularity—a quality metric measuring the density of intra-community edges relative to a random baseline—while incorporating refinement mechanisms to prevent the formation of poorly connected communities.

2. **Louvain Algorithm** [21]: An iterative modularity maximization approach that operates in two phases: local optimization, where nodes are reassigned to communities to maximize modularity gain, followed by aggregation, where communities are collapsed into super-nodes. This process repeats until no further improvement is possible.

3. **Label Propagation**: A computationally efficient network clustering algorithm in which each node iteratively adopts the most common label among its neighbors. Convergence typically occurs rapidly, making this approach suitable for large graphs, though the results may be non-deterministic.

For each algorithm, the system constructs an in-memory graph projection filtered by the current decomposition job context, executes the selected algorithm, and writes the resulting community assignments back to persistent storage. Each class node is annotated with multiple community identifiers—one per algorithm—enabling subsequent comparative analysis. The use of graph-native storage and analytics infrastructure allows the system to efficiently process codebases comprising thousands of classes and tens of thousands of dependencies. Furthermore, the modular design permits straightforward integration of additional clustering algorithms or custom partitioning heuristics.

4.1.4 Phase 4: Semantic Refinement via AI-Assisted Decomposition

The final phase applies semantic analysis to complement the purely structural clustering performed in the previous phase. While graph-based community detection identifies groups of classes with strong structural coupling, it operates without awareness of domain concepts, business capabilities, or actor interactions. To address this limitation, the workflow integrates large language model-based analysis to refine service boundaries according to domain-driven design (DDD) or actor-driven decomposition (ADD) principles.

The phase sequentially applies two decomposition strategies. **Domain-Driven Decomposition** identifies bounded contexts and business capabilities, grouping functionality into services with names reflecting domain concepts. **Actor-Driven Decomposition** identifies user roles

and workflows, grouping functionality according to which actors interact with it. Each strategy produces an independent set of service boundaries, enabling architects to compare structural, domain-driven, and actor-driven perspectives side by side.

The dependency graph is transformed into a structured representation and submitted to the language model with strategy-specific prompts. Responses are validated against an expected schema and persisted to storage, with class nodes updated to reflect the assigned service boundaries.

It is important to note that AI-generated decompositions are not perfect solutions. Language models lack deep domain expertise and contextual understanding of the specific business requirements, team structures, and operational constraints that influence real-world decomposition decisions. The generated candidates should therefore be viewed as a starting point for exploration rather than definitive recommendations, requiring validation and refinement by architects with domain knowledge.

4.2 Workflow Output and Visualization

Upon completion of the static code analysis workflow, ServiceSlicer produces a set of decomposition results that enable architects to explore and compare alternative microservice boundaries. For each analysis run, the system generates multiple decomposition candidates derived from both algorithmic clustering techniques (Louvain, Leiden, and Label Propagation) and semantically informed approaches based on Domain-Driven Design and Actor-Driven Design principles. Each candidate specifies proposed service boundaries by assigning classes to services, complemented by descriptive service names and quality metrics, allowing structural and semantic decompositions to be evaluated side by side.

To support intuitive analysis, ServiceSlicer provides an interactive visualization of the class-level dependency graph, as shown in Figure 4.2. Nodes in the graph represent classes and are colored according to their assigned service in the selected decomposition candidate, while directed edges capture dependencies such as method invocations and field accesses. Users can dynamically switch between

decomposition candidates and interact with the graph through zooming and node selection, making it possible to examine cohesion within services and dependencies that cross service boundaries. This visual representation complements quantitative metrics by making architectural trade-offs explicit and enabling early identification of potential integration hotspots before proceeding to implementation or performance evaluation.

Figure 4.2: Visualization of candidate microservice boundaries with color-coded clusters

5 Performance Benchmarking Workflow

This chapter operationalizes the Multi-Level Scalability Assessment methodology introduced in Section 2.4. The concepts defined there—baseline thresholds, domain metric, scalability footprint, scalability gap, and operational profiles—are implemented through ServiceSlicer's benchmarking infrastructure. This chapter describes how the platform automates deployment, workload generation, metrics collection, and multi-level analysis to make the methodology practical for iterative architectural evaluation.

5.1 Methodological Foundation

The benchmarking workflow implements the Multi-Level Scalability Assessment framework introduced by Avritzer et al. [9]. As described in Section 2.4, this framework compares architectural variants by measuring scalability at both system and operation levels, using baseline thresholds to determine pass/fail criteria and aggregating results into a domain metric.

The contribution of this thesis is not the scalability assessment methodology itself, but rather the implementation of an automated infrastructure that operationalizes this existing methodology within the context of monolith-to-microservices decomposition. Specifically, this work provides: (1) automated deployment and orchestration of architectural variants; (2) workload generation based on behavior models; (3) systematic collection and aggregation of performance metrics; and (4) integration with the static decomposition analysis workflow described in the previous chapter.

5.2 Workflow Overview

The benchmarking workflow is designed around the principle of comparative evaluation: it measures the performance of multiple architectural variants—typically a baseline monolithic system and one or more candidate decompositions—under identical workload conditions. The workflow is structured as a multi-phase process encompassing system

definition, workload specification, validation, execution, and results analysis. Each architectural variant is represented as an independent system under test (SUT), encapsulating its complete deployment configuration, runtime dependencies, and initialization requirements. These components are defined separately from the benchmark itself, enabling reuse across different experiments and facilitating systematic comparison of alternative architectural designs.

5.2.1 Phase 1: System Under Test Definition

The first phase establishes the architectural variants to be compared through the creation of system under test (SUT) definitions. Each SUT defines a complete architectural variant to be evaluated. An SUT includes a Docker Compose deployment descriptor specifying all services, resource allocations, and dependencies; operational metadata such as health-check endpoint, exposed port, and startup timeout; and optional database initialization configurations that seed persistent storage with representative application state prior to testing. This ensures each variant can be deployed, validated, tested, and torn down automatically and reproducibly, with consistent initial conditions across all benchmark runs.

5.2.2 Phase 2: Operational Setting Specification

An operational setting implements the operational profile concept defined in Section 2.4.5, providing the workload specification for benchmarking (see Appendix C, Figure C.2). The specification is grounded in an OpenAPI contract that defines all exposed operations. The operational setting comprises two components: behavior models and load level distribution.

Behavior Models

Behavior models define the sequences of API operations that actors execute. The system supports three specification modes:

- **Interactive Definition**: Models are constructed through a guided user interface for specifying operation sequences, parameter bindings, and timing constraints.

- **Structured Import**: Models are provided as JSON data that can be generated programmatically or exported from prior runs, enabling version control and automation.
- **AI-Assisted Synthesis**: Models can be synthesized automatically from the API contract using large language models when empirical usage data is unavailable.

Each behavior model specifies the actor type, relative frequency of occurrence, and the complete operation sequence including HTTP methods, endpoint paths, parameters, headers, and request bodies. Variable bindings allow values from one response to be injected into subsequent requests, enabling stateful workflows. Configurable think times between operations model realistic user pauses.

Load Level Distribution

The load level distribution specifies discrete concurrency levels and their probabilities, as required by the scalability assessment methodology. Users define the target number of concurrent virtual users for each level and assign probability weights that reflect expected production conditions.

5.2.3 Phase 3: Benchmark Definition

With the SUTs defined, along with a preconfigured operational setting, a benchmark instance is created to link these components and initiate the empirical evaluation process (see Appendix C, Figure C.3). A benchmark serves as a lightweight container that associates a specific baseline architecture, one or more target architectures, and an operational setting defining the workload to be applied. By separating the definition of architectural variants, workload models, and benchmark experiments, the system enables reuse of components across multiple studies and facilitates systematic exploration of alternative designs under varied operational scenarios.

5.2.4 Phase 4: Configuration Validation

Before executing a full benchmark, an optional validation phase verifies the correctness and consistency of all inputs associated with a

given SUT and operational setting. This step detects configuration errors early in the workflow, preventing wasted computational resources and ensuring that subsequent benchmark results are meaningful and reproducible.

The validation phase proceeds through several verification steps. First, the deployment configuration is validated syntactically and the SUT is deployed to confirm that all containers can be instantiated successfully. The health-check endpoint is probed repeatedly until the system signals readiness or a timeout occurs. If database seeding has been configured, the initialization scripts are executed to verify their compatibility with the deployed database schema and ensure that representative data can be loaded successfully.

To validate the operational setting, the system executes a diagnostic workload that invokes each operation defined in the behavior models exactly once, sequentially and without concurrency. This diagnostic run logs complete request and response data for every operation, enabling inspection of the actual API interactions and detection of errors such as incorrect variable bindings, malformed request payloads, or unexpected response structures. The diagnostic outputs are persisted for later inspection, providing traceability and supporting iterative refinement of behavior models.

By identifying misconfigurations prior to full-scale benchmarking, the validation phase serves as an important safeguard that enhances the reliability and efficiency of the experimental process. It ensures that measured performance differences reflect genuine architectural characteristics rather than artifacts of misconfigured workloads or deployment failures.

5.2.5 Phase 5: Benchmark Execution

Once configuration validation succeeds, the full execution phase performs automated, end-to-end performance testing of all architectural variants at every load level defined in the operational profile. The execution is structured as an iterative process that deploys, tests, and tears down each system configuration sequentially, ensuring consistent execution conditions and enabling precise comparative analysis across architectural variants.

Test Case Structure and Execution Plan

A benchmark run constructs a structured execution plan using a two-layer abstraction. A **TestCase** encapsulates a single test execution for a specific SUT at a particular load level, storing results, collected metrics, and execution state. A **TestSuite** groups all test cases for a single SUT across all load levels defined in the operational profile. The total number of test cases equals $|\text{SUTs}| \times |\text{load levels}|$, organized into $|\text{SUTs}|$ test suites.

When creating a benchmark, one SUT must be designated as the baseline. The baseline test suite's lowest load level establishes the scalability thresholds for each operation, as defined in Section 2.4.2. These thresholds serve as the performance reference for evaluating all architectures—including the baseline itself at higher loads.

The workflow executes test suites sequentially, starting with the baseline test suite. Within each suite, test cases execute sequentially in order of ascending load. Each test case progresses through well-defined states, with suite and benchmark status derived from the aggregate state of constituent tests. This structured approach ensures traceability, supports restart semantics in case of infrastructure failures, and enables incremental result inspection as tests complete.

Test Case Execution Procedure

Each test case follows a standardized execution procedure. The designated SUT is deployed and validated for operational readiness. The workload generator then executes the operational profile, spawning concurrent virtual users that repeatedly execute behavior models according to configured frequencies and load levels. Each operation invocation is instrumented to capture response times and invocation counts. Following test completion, performance metrics are computed from steady-state measurements, excluding ramp-up periods. Results are persisted and the SUT is systematically torn down to ensure isolation between test cases.

Performance Metric Computation Upon test completion, the framework computes aggregated performance statistics from steady-state measurements. Key metrics include mean response time and standard

deviation for each operation, invocation frequency across the workload, and pass/fail classification based on scalability thresholds. The platform then computes the multi-level scalability indicators defined in Section 2.4: domain metric, scalability footprint, scalability gap, and performance offset.

Result Persistence and System Teardown Computed metrics and test case metadata are persisted to the relational database. The SUT is then systematically torn down to ensure clean isolation between successive test cases.

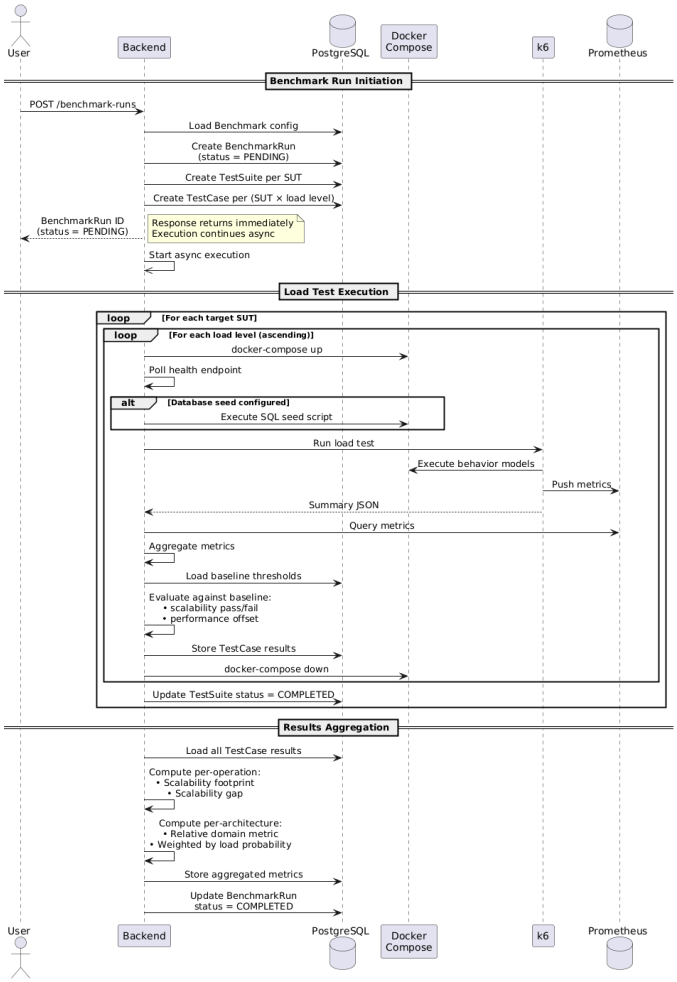

Figure 5.1: Benchmark Execution Sequence Diagram

6 Demonstration of ServiceSlicer in Practice

This chapter demonstrates the practical application of ServiceSlicer through two representative examples that illustrate how the platform supports monolith decomposition workflows. Rather than presenting a formal empirical evaluation with controlled experiments and statistical analysis, this chapter provides a hands-on preview of the tool's capabilities, showing how architects can use it to explore decomposition strategies and assess their viability through automated performance testing.

The demonstrations examine two systems in different contexts. The first is RealWorld Conduit, a medium-complexity demonstration application implementing a standardized blogging platform specification. This system serves as an accessible example that illustrates the complete workflow—from static analysis through decomposition candidate generation to performance benchmarking—in a well-understood domain. The second is Technika, an industrial fleet and equipment management system deployed in production. This example demonstrates how ServiceSlicer handles a real-world codebase with realistic usage patterns derived from actual system logs.

These demonstrations serve three purposes. First, they validate that ServiceSlicer can successfully process real applications, extract meaningful dependency structures, and execute automated benchmarks against containerized deployments. Second, they illustrate the types of insights the platform provides—including structural metrics, semantic decomposition candidates, and multi-level scalability assessments—and how these insights support architectural decision-making. Third, they reveal practical considerations that arise when applying the tool, such as the effort required to prepare operational profiles, the interpretability of clustering results, and the relationship between static analysis and runtime behavior.

The chapter is structured as follows. Section 6.1 presents the demonstration of ServiceSlicer applied to RealWorld, walking through the complete workflow from dependency analysis to performance comparison. Section 6.2 demonstrates the tool's application to Technika, a production system with realistic usage patterns. Section 6.3 synthesizes observations across both examples, discussing patterns and

practical implications. Finally, Section 6.4 reflects on the scope and limitations of these demonstrations.

6.1 Demonstration 1: RealWorld Conduit

6.1.1 System Overview

The **RealWorld** Conduit application is a medium-fidelity blogging platform implementing the RealWorld specification. This application provides a representative example of a typical web application architecture, featuring user management, content publishing, social interactions, and content discovery functionality. The system comprises approximately 5,000 lines of Java code organized into a monolithic Spring Boot application, with a RESTful API defined by an OpenAPI contract. The application interacts with a PostgreSQL database for persistent storage and includes features such as user authentication, article management, commenting, and favoriting.

6.1.2 Static Analysis

The static analysis of the RealWorld Conduit application resulted in a dependency graph comprising 81 nodes and 183 edges. Given the limited size of the codebase, the complete decomposition job completed in 58 seconds, demonstrating low computational overhead for static analysis on small to medium systems. Both the Louvain and Leiden community detection algorithms produced comparable decompositions consisting of five clusters. The corresponding modularity scores of 0.56 for Louvain and 0.55 for Leiden indicate a clearly identifiable, though not extreme, community structure within the dependency graph, suggesting the presence of cohesive groups of classes with relatively fewer dependencies across cluster boundaries.

In contrast, the Label Propagation algorithm identified only two clusters, with most classes grouped into a single large cluster. This difference is expected: Label Propagation tends to produce coarser partitions in well-connected graphs because labels spread easily across the network. The modularity-based algorithms (Louvain and Leiden) actively optimize for balanced partitions with fewer cross-cluster edges, enabling them to detect finer-grained structure even when the overall

Figure 6.1: Leiden algorithm decomposition of the RealWorld Conduit application

graph is densely connected. Figure 6.1 illustrates the Leiden decomposition, which we use as the basis for subsequent decomposition.

6.1.3 Generated Decomposition Candidates and Implementation

The Leiden algorithm's output was combined with AI-assisted decomposition suggestions and domain knowledge of the blogging platform to design a deployable microservices architecture. Based on this analysis, three microservices were extracted: an *article-service*, a *comments-service*, and a *user-service*. This decomposition aligns with the natural domain boundaries of a blogging platform, where articles, comments, and user management represent distinct areas of functionality with clear responsibilities and data ownership.

The services are deployed behind an API gateway that exposes the same REST API as the original monolith, ensuring functional equiva-

lence and enabling fair performance comparison. Each microservice manages its own database, enforcing data ownership and eliminating direct cross-service database access.

The microservice-based application was containerized using Docker Compose to support automated deployment during benchmarking. Representative seed data was generated to ensure that load tests run against a realistic dataset rather than an empty state.

6.1.4 Operational Setting

The operational setting was derived by recording network requests from representative user sessions on the deployed demo application. The list of API operations was extracted from the application's OpenAPI specification, ensuring complete coverage of the exposed endpoints. Two actor types were identified: *reader* and *writer*, refined into three behavior models. The *reader-passive* behavior represents passive content consumption. The *reader-active* behavior extends this with interactive actions such as posting comments and following authors. The *writer* behavior captures content creation activities. Table 6.1 lists the API operations used, and Table 6.2 shows the operation sequences for each behavior model.

Table 6.1: Operations used in the RealWorld Conduit load testing

Op.	OperationId	HTTP	Path
o1	getArticles	GET	/articles
o2	getTags	GET	/tags
o3	article	GET	/articles/{slug}
o4	getComments	GET	/articles/{slug}/comments
o5	userLogin	POST	/users/login
o6	createComment	POST	/articles/{slug}/comments
o7	follow	POST	/profiles/{username}/follow
o8	favoriteArticle	POST	/articles/{slug}/favorite
o9	createArticle	POST	/articles
o10	getFeed	GET	/articles/feed

To simulate realistic user behavior, think times were configured between operations. Two categories of delays were used: *automatic*

think times (1–200ms) representing browser processing and network latency between consecutive requests, and *user* think times (1–7 seconds) representing manual interactions such as reading content or clicking links. Content creation actions required longer delays—for example, a think time exceeding 15 seconds was configured before `createArticle` to simulate the time spent composing a blog post.

Table 6.2: Behavior models and operation sequences used in the Real-World Conduit load testing

Actor	Behavior model	Behavior mix	Steps (operation shortcuts)
Reader	reader-passive	0.5	o1 → o2 → o3 → o4 → o1 → o2 → o3 → o4
Reader	reader-active	0.3	o1 → o3 → o5 → o10 → o2 → o1 → o2 → o3 → o4 → o7 → o8 → o6
Writer	writer	0.2	o5 → o10 → o2 → o9 → o3 → o4

To determine appropriate load levels, preliminary tests were conducted to identify the system's capacity threshold—the point at which response times began to degrade significantly. Based on these observations, seven discrete load levels were defined ranging from 50 to 350 concurrent virtual users, with probability weights assigned to emphasize mid-range loads where the system operates under realistic stress. Table 6.3 summarizes the operational profile used for benchmarking.

6.1.5 Benchmark Execution and Results

The benchmarking workflow was executed following the procedure outlined in Chapter 5. Test cases were executed with a 30-second warm-up period (during which metrics were excluded) followed by a 5-minute measurement phase at each load level. Each test case was repeated three times to reduce variance from random fluctuations and transient system states. Both architectural variants were evaluated across all seven load levels defined in the operational profile.

Table 6.3: Operational profile load levels for RealWorld Conduit benchmarking

Concurrent Users	Probability
50	0.119
100	0.139
150	0.171
200	0.203
250	0.162
300	0.114
350	0.092

System-Level Results Figure 6.2 visualizes the relative domain metrics for both architectures at each load. The monolithic architecture achieved a total domain metric of 0.788, while the microservices architecture scored 0.600. The monolithic architecture demonstrated superior overall scalability, maintaining acceptable performance until 200 concurrent users before degradation became significant. The microservices variant began showing performance degradation earlier, at 150 concurrent users. This difference is reflected in the 18.8% lower domain metric for the decomposed architecture.

Operation-Level Results Figure 6.3 visualizes the scalability footprint for each operation across both architectures. The footprint indicates the maximum load level at which an operation consistently met its performance threshold. In the monolith, most operations sustained load up to 250 VUs, with `createComment` reaching 300 VUs and `userLogin` being the weakest at 200 VUs. The microservices architecture shows a more varied pattern: some operations degraded earlier while others improved.

Tables 6.4 and 6.5 present the detailed operation-level metrics for both architectures, including request counts, failure rates, scalability footprints, gaps, and performance offsets.

6.1.6 Observations and Insights

Monolith Outperformed Microservices in Overall Scalability The monolithic architecture achieved a domain metric of 0.788 compared

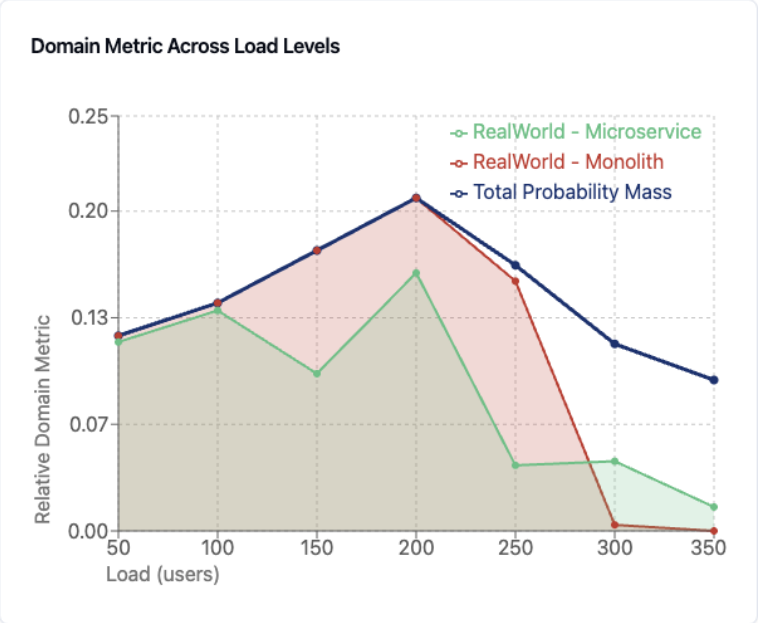

Figure 6.2: Domain metric comparison between Monolith and Microservices architectures for RealWorld Conduit

to 0.600 for the microservices variant—a 18.8% difference. The monolith sustained acceptable performance up to 200 concurrent users, while the microservices architecture began degrading at 150 users. This result contradicts the common assumption that microservices inherently improve scalability; for this application size and workload profile, the decomposition introduced overhead that outweighed any benefits from service isolation.

Inter-Service Communication Overhead The `getArticles` operation demonstrated the clearest evidence of decomposition overhead: its scalability footprint dropped from 250 concurrent users in the monolith to just 100 in the microservices variant. Since article retrieval in the decomposed architecture requires coordination across services, the additional network hops and serialization costs degraded performance substantially. This pattern highlights that read-heavy operations spanning multiple services are particularly vulnerable to distributed system overhead.

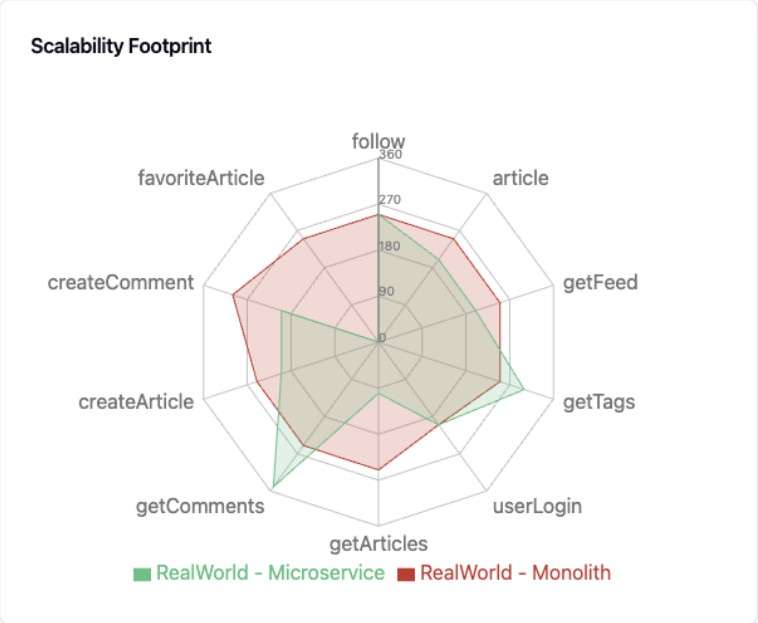

Figure 6.3: Operation-level scalability footprints for RealWorld Conduit (Monolith vs. Microservices)

Service Isolation Benefited Some Operations Not all operations suffered from decomposition. The `getTags` operation actually improved in the microservices architecture, sustaining load up to 300 VUs (compared to 250 VUs in the monolith) with a dramatically lower performance offset (0.413 vs. 5.504). Isolating tag management into a dedicated service reduced contention with other components, demonstrating that decomposition can benefit operations with minimal cross-service dependencies.

Authentication Remained a Bottleneck Across Both Architectures The `userLogin` operation exhibited early threshold violations in both architectures, with a footprint of 200 VUs. However, the microservices variant showed significantly worse degradation severity (performance offset of 5.868 vs. 0.151). This suggests that authentication represents a shared bottleneck—likely due to database contention or connection pool exhaustion—that decomposition alone cannot resolve and may even make worse.

Table 6.4: Operation-level scalability metrics for RealWorld Conduit (Monolith)

Operation	Requests	Failed	Footprint	Gap	Offset
follow	6,952	0	250	0.032	0.064
article	38,222	0	250	0.207	21.213
getFeed	11,567	0	250	0.054	5.340
getTags	42,087	0	250	0.205	5.504
userLogin	11,703	136	200	0.060	0.151
getArticles	37,549	0	250	0.186	7.075
getComments	31,770	0	250	0.173	2.544
createArticle	4,615	1,111	250	0.022	7.805
createComment	6,655	0	300	0.035	1.872
favoriteArticle	6,612	2	250	0.032	2.279

Write Operation Failures The `createArticle` operation experienced high failure rates in both architectures: 1,111 failures out of 4,615 requests in the monolith, and 1,080 out of 4,242 in microservices. Write operations under load appear particularly vulnerable to timeouts and resource contention, and distributing them across services did not improve reliability.

Decomposition Introduced New Failure Modes Some operations that were reliable in the monolith became problematic after decomposition. The `getComments` operation had zero failures in the monolith but experienced 1,084 failures in the microservices variant. Similarly, `favoriteArticle` failed to meet its performance threshold at any load level in the microservices architecture, preventing meaningful scalability metrics from being computed. These new failure modes illustrate that decomposition can introduce reliability risks not present in the original system.

Static Analysis Could Not Predict Runtime Coupling The static decomposition analysis identified structurally coherent clusters, but could not reveal the runtime coupling that emerged under load. Operations like `article`, `getArticles`, and `getFeed` exhibited correlated degradation patterns, indicating synchronous inter-service dependencies that only became apparent during benchmarking. This finding

Table 6.5: Operation-level scalability metrics for RealWorld Conduit (Microservices)

Operation	Requests	Failed	Footprint	Gap	Offset
follow	6,036	0	250	0.036	0.057
article	34,073	0	200	0.194	1.502
getFeed	10,724	0	200	0.066	2.446
getTags	38,240	0	300	0.212	0.413
userLogin	10,724	0	200	0.066	5.868
getArticles	33,998	0	100	0.192	0.113
getComments	28,212	1,084	350	N/A	N/A
createArticle	4,242	1,080	200	0.026	1.161
createComment	5,829	0	200	0.028	0.779
favoriteArticle	5,802	5	N/A	N/A	N/A

validates the core thesis motivation: static analysis alone is insufficient for evaluating decomposition quality.

Granular Metrics Enabled Targeted Optimization Insights The multi-level assessment methodology provided actionable insights that aggregate metrics would obscure. While the `article` operation had a moderate scalability gap (0.207), its extreme performance offset (21.213 in the monolith) revealed that threshold violations were severe when they occurred. Conversely, `getTags` had a similar gap (0.205) but much lower offset (5.504), indicating more gradual degradation. These distinctions help architects prioritize optimization efforts based on both frequency and severity of performance violations.

6.2 Demonstration 2: Technika

6.2.1 System Overview

Technika is a fleet and equipment management system designed for tracking machinery and vehicle usage, employee work hours, and project-based resource allocation. The application supports core business workflows including device borrowing and returning, time tracking, usage-based billing, and audit trail management. The system is implemented as a Kotlin-based Spring Boot monolith, managing five

primary domain entities: devices (equipment and vehicles with pricing models), employees (workers who borrow devices and log hours), projects (work assignments), transactions (device usage records tracking kilometers, engine hours, fuel consumption, and costs), and work logs (time entries linking employees to projects). The application exposes a RESTful API and relies on PostgreSQL for persistent storage, providing typical enterprise features such as resource management, usage tracking, and project cost calculation.

As a production system with real operational history, Technika provides an opportunity to demonstrate ServiceSlicer's applicability to industrial contexts where deployment decisions carry genuine business consequences and where realistic usage patterns can be derived from actual system logs and domain expertise.

6.2.2 Static Analysis Workflow

The Technika monolith was uploaded to ServiceSlicer as a compiled JAR artifact and processed through the static analysis workflow described in Chapter 4. The dependency extraction phase identified the application's structural composition, producing a dependency graph consisting of 60 class nodes connected by 153 directed edges representing method invocations, field accesses, and type dependencies.

Following graph construction, three community detection algorithms were applied to identify candidate service boundaries. The Louvain algorithm achieved a modularity score of 0.4554, indicating moderate structural separation between detected clusters. The Leiden algorithm produced a slightly higher modularity of 0.4775, suggesting improved cluster quality through its refinement mechanisms. The Label Propagation algorithm converged to a 3-cluster partition, reflecting a coarser-grained decomposition compared to the hierarchical methods.

These modularity values, while lower than those typically observed in larger systems, are characteristic of smaller monoliths where tight integration across functional areas is common. The relatively compact codebase and limited number of domain entities naturally constrain the degree of structural separation that can be achieved through purely algorithmic partitioning.

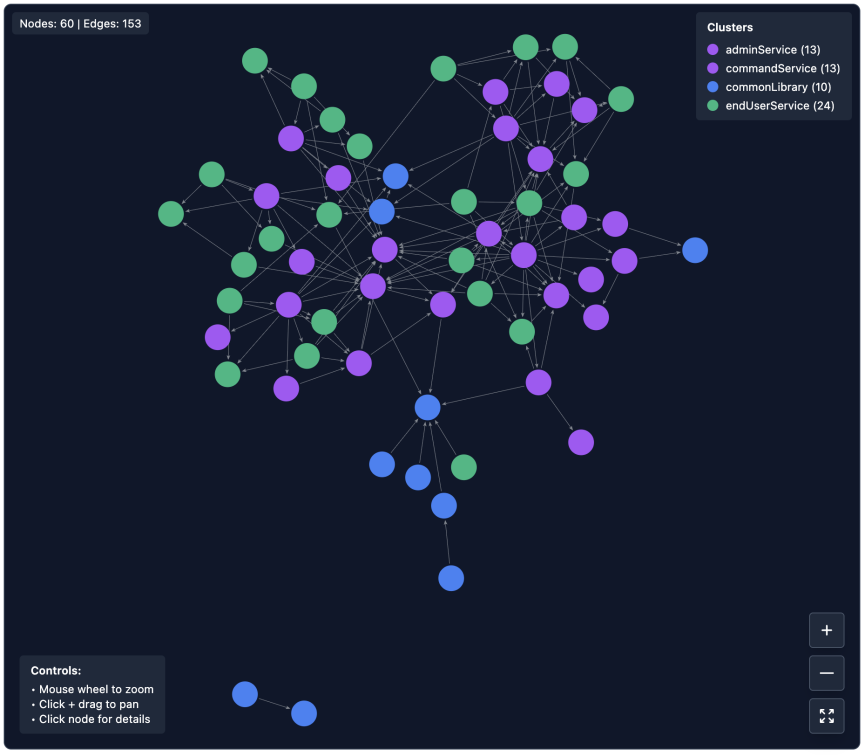

Figure 6.4: AI-assisted Actor-Driven decomposition graph of the Technika application

6.2.3 Generated Decomposition Candidates and Implementation

The algorithmic clustering results served as initial guidance, but the final decomposition strategy was developed with a domain expert who understood the system's operational context. Rather than pursuing fine-grained decomposition based solely on structural clustering, two coarser-grained services were extracted that aligned with the primary actor types:

- **Admin Service**: Manages master data synchronization with external systems, handling devices, employees, and projects. Used primarily by automated processes for data updates.
- **Worker Service**: Handles operational workflows performed by field workers, including transaction management (device usage,

fuel consumption, mileage) and work log entries. Supports the high-frequency operations that constitute the majority of daily system usage.

Each microservice was containerized using Docker Compose. To ensure realistic testing conditions, the benchmark environment was initialized using a database dump from the production server, providing representative data volume and distribution patterns.

6.2.4 Operational Profile Configuration

The operational setting for Technika was derived from analysis of real system usage patterns conducted in collaboration with a domain expert. Building on the actor-aligned decomposition described in the previous section, behavior models were constructed to capture the typical operation sequences executed by the Admin Actor and Worker Actor.

Table 6.6: Operations used in the Technika load testing

Op.	OperationId	HTTP	Path
o1	listAll_2	GET	/devices
o2	save_2	PUT	/devices/{id}
o3	listAll_1	GET	/employees
o4	save_1	PUT	/employees/{id}
o5	listAll	GET	/projects
o6	save	PUT	/projects/{id}
o7	listSinceLastUpdate	GET	/work-logs/since
o8	listSinceLastUpdate_1	GET	/transactions/since
o9	getCurrentServerTime	GET	/server-time
o10	save_3	POST	/work-logs
o11	save_4	POST	/transactions

The behavior mix was weighted according to observed usage frequencies: worker operations constitute 90% of system traffic, while external system synchronization occurs at 10% frequency.

Each step in the behavior models includes a configurable think time to simulate realistic interaction delays. Automated operations (such

Table 6.7: Behavior models and operation sequences used in the Technika load testing

Actor	Behavior model	Behavior mix	Operations
Admin	sync-data-write	0.05	o1 → o2 → o2 → o2 → o3 → o4 → o4 → o4 → o5 → o6 → o6 → o6
Admin	sync-data-read	0.05	o7 → o8
Worker	create-worklog	0.3	o3 → o5 → o9 → o10
Worker	create-transaction	0.6	o1 → o5 → o3 → o9 → o11 → o11

as data synchronization) use short delays of 1–200ms, while human-operated steps (such as form submissions) use longer delays of 1000–5000ms to reflect typical user pauses during interactive workflows.

Load levels were specified to reflect realistic operational conditions, ranging from 20 to 70 concurrent users distributed across the two actor types. This range encompasses both typical daily usage patterns and peak load scenarios observed during periods of high field activity.

The operational profile was validated through diagnostic runs to confirm correct API interaction patterns, verify data dependencies between sequential operations, and ensure that the modeled workload accurately represented production usage characteristics.

6.2.5 Benchmark Execution and Results

The benchmarking workflow was executed following the same procedure as for RealWorld Conduit. Test cases were configured with a 30-second warm-up period followed by a 5-minute measurement phase at each load level. Each test case was repeated three times to reduce variance. Both architectural variants—the original monolith and the two-service microservices decomposition—were evaluated across all load levels defined in the operational profile.

System-Level Results Figure 6.5 visualizes the domain metrics for both architectures across load levels. The monolithic architecture achieved a total domain metric of 0.812, while the microservices ar-

chitecture scored 0.739. Unlike the RealWorld demonstration where microservices performed significantly worse, the Technika decomposition maintained relatively competitive performance—only a 7.3% reduction in the domain metric.

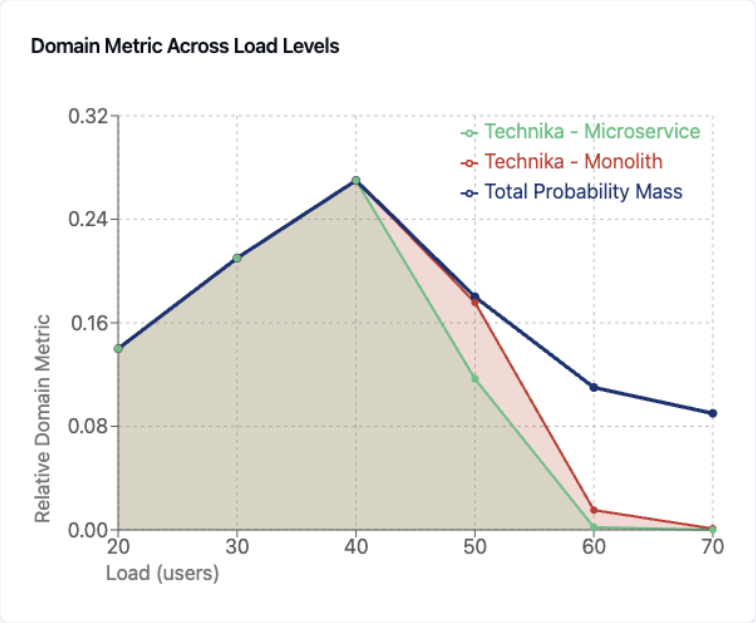

Figure 6.5: Domain metric comparison between Monolith and Microservices architectures for Technika

Operation-Level Results Figure 6.6 visualizes the scalability footprint for each operation across both architectures. In the monolith, most operations sustained load between 50 and 70 concurrent users, with `save` (projects) being the weakest at 40 VUs. The microservices architecture shows a consistent pattern of slightly earlier degradation across write operations.

Tables 6.8 and 6.9 present the detailed operation-level metrics for both architectures, including request counts, failure rates, scalability footprints, gaps, and performance offsets.

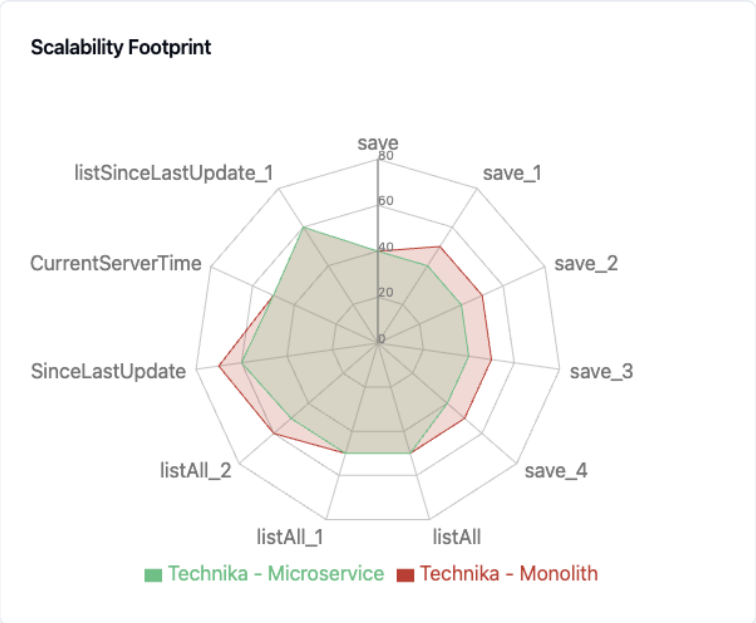

Figure 6.6: Operation-level scalability footprints for Technika (Monolith vs. Microservices)

6.2.6 Observations and Insights

Smaller Performance Gap Than RealWorld The Technika decomposition showed a much smaller performance difference between architectures compared to RealWorld Conduit. The microservice architecture achieved only a 7.3% reduction, compared to 18.8% in the RealWorld experiment. This suggests that the actor-aligned decomposition strategy, which produced two coarse-grained services with minimal inter-service communication, preserved more of the monolith's performance characteristics.

Write Operations Degraded Uniformly All write operations (`save`, `save_1`, `save_2`, `save_3`, `save_4`) showed reduced scalability footprints in the microservices variant, dropping from 40–50 VUs to a uniform 40 VUs. This consistent degradation pattern differs from RealWorld, where write operation behavior varied significantly. The uniformity

Table 6.8: Operation-level scalability metrics for Technika (Monolith)

Operation	Requests	Failed	Footprint	Gap	Offset
save	1,341	0	40	0.024	0.040
save_1	1,335	0	50	0.021	0.465
save_2	1,333	0	50	0.021	0.445
save_3	3,165	0	50	0.054	0.426
save_4	13,186	0	50	0.227	0.422
listAll	10,238	0	50	0.175	0.095
listAll_1	10,238	0	50	0.175	0.083
listAll_2	7,073	0	60	0.120	0.526
listSinceLastUpdate	571	0	70	N/A	N/A
getCurrentServerTime	9,758	0	50	0.167	0.271
listSinceLastUpdate_1	571	0	60	0.011	0.352

suggests that the Admin Service and Worker Service share similar write path characteristics.

Read Operations Remained Stable The read operations (`listAll`, `listAll_1`, `listAll_2`) maintained identical scalability footprints (50–60 VUs) across both architectures. This stability indicates that the decomposition did not introduce significant overhead for read-heavy operations—a notable contrast to RealWorld where `getArticles` dropped from 250 to 100 VUs.

No Failed Requests in Either Architecture Unlike RealWorld Conduit, which experienced over 1,000 failed requests in both architectures, Technika processed all requests successfully. This reliability difference likely reflects Technika's maturity as a production application that has been fine-tuned for stability under real-world usage conditions.

Lower Performance Offsets in Microservices Several operations exhibited lower performance offsets in the microservices architecture. For example, `save_4` (transaction creation) had an offset of 0.422 in the monolith but only 0.156 in microservices, and `listAll_2` dropped from 0.526 to 0.109. This counterintuitive result suggests that isolating the Worker Service reduced contention for high-frequency transaction operations.

Table 6.9: Operation-level scalability metrics for Technika (Microservices)

Operation	Requests	Failed	Footprint	Gap	Offset
save	1,538	0	40	0.026	0.169
save_1	1,529	0	40	0.025	0.138
save_2	1,515	0	40	0.025	0.178
save_3	3,095	0	40	0.050	0.134
save_4	12,706	0	40	0.226	0.156
listAll	10,002	0	50	0.174	0.433
listAll_1	10,002	0	50	0.174	0.363
listAll_2	6,907	0	50	0.119	0.109
listSinceLastUpdate	535	0	60	0.010	0.027
getCurrentServerTime	9,448	0	50	0.164	0.486
listSinceLastUpdate_1	535	0	60	0.010	0.434

Actor-Aligned Decomposition Minimized Cross-Service Calls The two-service decomposition aligned with actor boundaries (Admin vs. Worker) resulted in minimal inter-service communication during typical workflows. Each behavior model's operations were largely contained within a single service, avoiding the cascading latency observed in RealWorld where article retrieval required coordination across multiple services.

6.3 Scope and Representativeness

The two demonstrated systems represent different points on the complexity spectrum but share certain characteristics that may limit generalizability. Both are relatively small (fewer than 100 classes), built on similar technology stacks (Spring Boot, PostgreSQL), and follow conventional layered architectures. They do not exhibit the extreme coupling, legacy complexity, or polyglot characteristics typical of large enterprise monoliths. Furthermore, single-host benchmarking cannot capture the network latency and distributed coordination overhead that emerge in production-scale deployments. These demonstrations validate that ServiceSlicer functions correctly on representative applications, but additional case studies on larger and more diverse systems would strengthen confidence in the tool's broader applicability.

6.4 Summary

The demonstrations presented in this chapter validate that ServiceSlicer can successfully process real applications, generate meaningful decomposition candidates, and execute automated benchmarks against containerized deployments. Both RealWorld Conduit and Technika progressed through the complete workflow, producing actionable insights for architectural decision-making.

The two demonstrations yielded contrasting results. RealWorld Conduit's three-service decomposition resulted in a 24% domain metric reduction with significant inter-service overhead, while Technika's actor-aligned two-service decomposition showed only a 9% reduction. In both cases, the monolithic architecture outperformed microservices, reinforcing that decomposition decisions should be grounded in empirical performance data rather than assumptions about microservices benefits.

7 Conclusion

ServiceSlicer addresses a critical gap identified in recent systematic reviews [5]: the lack of integrated tooling that spans from boundary identification through deployment-based validation. While existing approaches typically focus on either static analysis or runtime evaluation in isolation, ServiceSlicer combines both perspectives, enabling architects to ground architectural decisions in empirical performance data.

7.1 Summary of Contributions

This thesis makes three primary contributions:

Integrated Decomposition Platform. ServiceSlicer combines static analysis and performance benchmarking into a unified workflow. The platform extracts dependencies from Java bytecode, applies graph-based clustering algorithms to identify candidate service boundaries, and supports AI-assisted semantic refinement. For validation, it automates deployment and load testing of architectural variants.

Automated Multi-Level Scalability Assessment. The platform provides an automated implementation of the Multi-Level Scalability Assessment methodology [9]. It automates deployment orchestration, executes workloads based on behavior models, and computes scalability metrics at both system and operation levels. This automation reduces evaluation effort and enables reproducible performance comparison across architectural variants.

Empirical Validation. The platform was applied to two applications: RealWorld Conduit and Technika. These demonstrations validate that ServiceSlicer can process actual codebases, generate meaningful decomposition candidates, and execute automated benchmarks against containerized deployments.

7.2 Key Findings

The demonstrations revealed several insights relevant to practitioners considering monolith decomposition:

Static analysis provides guidance, not prescriptions. Clustering algorithms produce structurally coherent partitions, but these require interpretation through domain knowledge to yield practically deployable microservices.

Coarse-grained decompositions can outperform fine-grained ones. The actor-aligned two-service decomposition of Technika showed only 9% performance degradation, compared to 24% for RealWorld's three-service decomposition. Organizing services around actor boundaries rather than pursuing maximal structural granularity preserved more of the monolith's performance characteristics.

Deployment-based validation reveals hidden bottlenecks. Automated benchmarking uncovered performance characteristics unpredictable from static analysis alone—including inter-service communication overhead, contention patterns, and cascading degradation under load.

Microservices do not inherently improve scalability. In both demonstrations, the monolithic architecture outperformed the decomposed variant, reinforcing that decomposition decisions should be grounded in empirical data rather than assumptions.

7.3 Future Directions

Several avenues exist for future enhancement:

Multi-language static analysis. Extend dependency extraction to support additional programming languages. This would broaden the platform's applicability, as the benchmarking workflow already supports any language through containerized deployments and REST APIs.

Scalable graph processing. Introduce asynchronous transactions and selective fetching to handle larger codebases with tens of thousands of classes more efficiently.

Advanced decomposition algorithms. Integrate additional clustering approaches such as spectral clustering or hierarchical methods, and provide interactive refinement capabilities.

Workflow robustness. Support stopping running jobs, restarting individual test cases, and recovering from partial failures more gracefully.

Benchmark reporting. Generate exportable summaries of benchmark results in PDF or structured data formats suitable for documentation and stakeholder communication.

A REST API Reference

This appendix provides a reference of the REST API endpoints exposed by the ServiceSlicer backend. The API follows RESTful conventions and uses JSON for request and response payloads. Complete OpenAPI documentation is available at `/swagger-ui.html` when the backend is running.

A.1 Decomposition Jobs

Endpoints for managing static code analysis jobs that extract dependencies and identify candidate microservice boundaries.

Table A.1: Decomposition Job Endpoints

Method	Path	Description
GET	/decomposition-jobs	List all jobs with pagination
GET	/decomposition-jobs/{id}	Get job details with candidates and metrics
POST	/decomposition-jobs	Create a new job from uploaded JAR
POST	/decomposition-jobs/{id}/restart	Restart a failed job

A.2 Systems Under Test

Endpoints for managing deployable system configurations used in benchmarking.

Table A.2: System Under Test Endpoints

Method	Path	Description
GET	/systems-under-test	List all SUTs with pagination
GET	/systems-under-test/{id}	Get SUT details with Docker config
POST	/systems-under-test	Create a new SUT
PUT	/systems-under-test/{id}	Update an existing SUT
DELETE	/systems-under-test/{id}	Delete a SUT

A.3 Operational Settings

Endpoints for managing workload specifications including behavior models and operational profiles.

Table A.3: Operational Setting Endpoints

Method	Path	Description
GET	/operational-settings	List all settings with pagination
GET	/operational-settings/{id}	Get setting details
GET	/operational-settings/{id}/ usage-profile	Get behavior models
POST	/operational-settings	Create a new setting
PUT	/operational-settings/{id}	Update an existing setting
DELETE	/operational-settings/{id}	Delete a setting

A.4 Benchmarks

Endpoints for managing benchmark definitions that link SUTs with operational settings.

Table A.4: Benchmark Endpoints

Method	Path	Description
GET	/benchmarks	List all benchmarks with pagination
GET	/benchmarks/{id}	Get benchmark details with test suites
POST	/benchmarks	Create a new benchmark
PUT	/benchmarks/{id}	Update an existing benchmark
POST	/benchmarks/{id}/generate-bm	Generate behavior models using AI
POST	/benchmarks/{id}/sut/ {sutId}/validate	Validate SUT configuration

A.5 Benchmark Runs

Endpoints for executing and monitoring benchmark runs.

Table A.5: Benchmark Run Endpoints

Method	Path	Description
GET	/benchmark-runs	List runs for a benchmark
GET	/benchmark-runs/{id}	Get run details with test results
GET	/benchmark-runs/{id}/ usage-profile	Get behavior models used in run
POST	/benchmark-runs	Create and start a new run
POST	/benchmark-runs/{id}/restart	Restart a failed run

A.6 Files

Endpoints for managing file uploads using a three-step upload flow (initiate, upload to presigned URL, complete).

Table A.6: File Endpoints

Method	Path	Description
GET	/files	List all uploaded files with pagination
GET	/files/{id}/download	Get a presigned download URL
POST	/files	Initiate upload, returns presigned URL
POST	/files/{id}/complete	Mark file upload as complete

A.7 API Operations

Endpoints for parsing and managing API operations extracted from OpenAPI specifications.

Table A.7: API Operation Endpoints

Method	Path	Description
GET	/api-operations	List operations for an OpenAPI file
POST	/api-operations	Parse and create operations from OpenAPI

B Deployment and Configuration

This appendix provides instructions for deploying and running the ServiceSlicer platform locally. The complete source code is available in the accompanying repository.

B.1 Prerequisites

The following software must be installed:

- **Docker** and **Docker Compose** — for running infrastructure services
- **Java 21** or later — for running the backend
- **Maven 3.9+** — for building the backend
- **Node.js 18+** and **npm** — for running the frontend

B.2 Infrastructure Services

ServiceSlicer requires four infrastructure services, configured via Docker Compose. The following listing shows the complete `compose.yaml` file located in the `backend/` directory:

Listing B.1: Docker Compose configuration (backend/compose.yaml)

```
services:
  postgres:
    image: 'postgres:17.4-alpine'
    environment:
      - 'POSTGRES_DB=serviceslicer'
      - 'POSTGRES_PASSWORD=postgres'
      - 'POSTGRES_USER=postgres'
    ports:
      - '33771:5432'
    volumes:
      - postgres-data:/var/lib/postgresql/data
```

```
neo4j:
  image: 'neo4j:5.26.13-community'
  environment:
    NEO4J_AUTH: neo4j/password
    NEO4J_PLUGINS: '["apoc", "graph-data-
       science"]'
    NEO4J_dbms_security_procedures_unrestricted
       : "gds.*,apoc.*"
    NEO4J_dbms_security_procedures_allowlist:
       "gds.*,apoc.*"
    NEO4J_apoc_export_file_enabled: "true"
    NEO4J_apoc_import_file_enabled: "true"
  ports:
    - '7474:7474'
    - '7687:7687'
  volumes:
    - neo4j-data:/data

minio:
  image: 'minio/minio:latest'
  command: server /data --console-address "
     :9001"
  environment:
    MINIO_ROOT_USER: minioadmin
    MINIO_ROOT_PASSWORD: minioadmin
  ports:
    - '9000:9000'
    - '9001:9001'
  volumes:
    - minio-data:/data

prometheus:
  image: 'prom/prometheus:v3.0.1'
  ports:
    - '9091:9090'
  volumes:
```

```
      - ./prometheus.yml:/etc/prometheus/
        prometheus.yml
      - prometheus-data:/prometheus
    command:
      - '--config.file=/etc/prometheus/
        prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--enable-feature=native-histograms'
      - '--web.enable-remote-write-receiver'
    extra_hosts:
      - "host.docker.internal:host-gateway"

volumes:
  postgres-data:
  neo4j-data:
  minio-data:
  prometheus-data:
```

The services provide the following functionality:

- **PostgreSQL** (port 33771): Stores domain entities, job metadata, and benchmark results
- **Neo4j** (ports 7474, 7687): Stores dependency graphs and executes community detection algorithms via the Graph Data Science plugin
- **MinIO** (ports 9000, 9001): S3-compatible object storage for JAR files, Docker Compose configurations, and other artifacts
- **Prometheus** (port 9091): Collects and stores performance metrics from k6 load tests

B.3 Running the Application

B.3.1 Starting the Backend

From the `backend/` directory, run:

```
mvn spring-boot:run
```

This command automatically starts the Docker Compose services (if not already running) and launches the Spring Boot application. The backend API will be available at `http://localhost:8080`. API documentation is accessible at `http://localhost:8080/swagger-ui.html`.

B.3.2 Starting the Frontend

From the `frontend/` directory, run:

```
npm install
npm run generate:api
npm run dev
```

The `generate:api` command generates TypeScript API client code from the backend's OpenAPI specification (requires the backend to be running). The frontend development server will be available at `http://localhost:3000`.

B.4 Service Endpoints Summary

Table B.1: Service endpoints after deployment

Service	URL	Purpose
Frontend	http://localhost:3000	Web UI
Backend API	http://localhost:8080	REST API
Swagger UI	http://localhost:8080/swagger-ui.html	API docs
Neo4j Browser	http://localhost:7474	Graph DB UI
MinIO Console	http://localhost:9001	Object storage UI
Prometheus	http://localhost:9091	Metrics

C User Interface Guide

This appendix presents the key user interface screens of the ServiceSlicer platform. The web-based interface provides access to both the static analysis and benchmarking workflows, enabling users to configure analyses, monitor progress, and inspect results through an intuitive graphical environment.

C.1 Decomposition Job Details

The decomposition job details page (Figure C.1) serves as the primary interface for exploring static analysis results. After a decomposition job completes, this page presents the extracted dependency graph alongside the generated decomposition candidates.

The main area displays an interactive visualization of the class-level dependency graph. Nodes represent Java classes extracted from the uploaded JAR file, while directed edges indicate structural dependencies such as method invocations, field accesses, and type references. Users can pan, zoom, and select individual nodes to inspect class details and their connections.

A selection control allows users to switch between different decomposition candidates. When a candidate is selected, nodes are colored according to their assigned service, making it easy to visually assess cluster cohesion and identify dependencies that cross service boundaries.

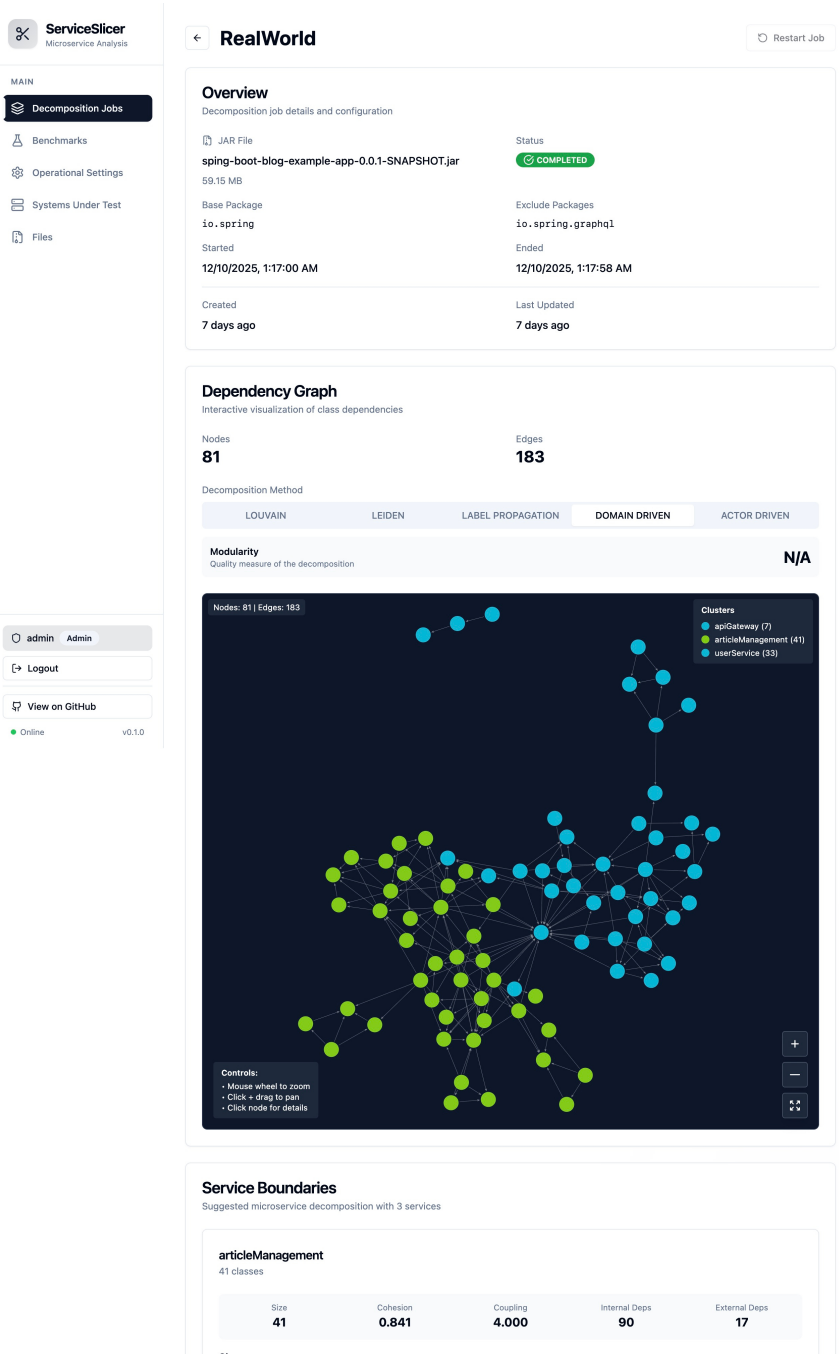

Figure C.1: Decomposition job details page showing the interactive dependency graph visualization with color-coded service assignments

C.2 Operational Setting Configuration

The operational setting configuration page (Figure C.2) enables users to define workload specifications for benchmarking. An operational setting captures the expected usage patterns of a system through behavior models and load level distributions.

The configuration process begins with uploading an OpenAPI specification that describes the API of the system under test. ServiceSlicer parses this specification to extract available operations, which then serve as building blocks for constructing behavior models.

The behavior model editor allows users to define sequences of API operations that represent realistic user workflows. For each step in a behavior model, users specify the operation to invoke, parameter values or variable bindings, request headers, and think time delays. Variable bindings enable stateful workflows where values extracted from one response (such as authentication tokens or resource identifiers) are injected into subsequent requests.

The operational profile section defines load levels and their probability distribution. Users specify discrete concurrency levels representing the number of simultaneous virtual users, along with the probability that each level occurs under realistic conditions. This probabilistic characterization ensures that scalability metrics emphasize commonly encountered load scenarios.

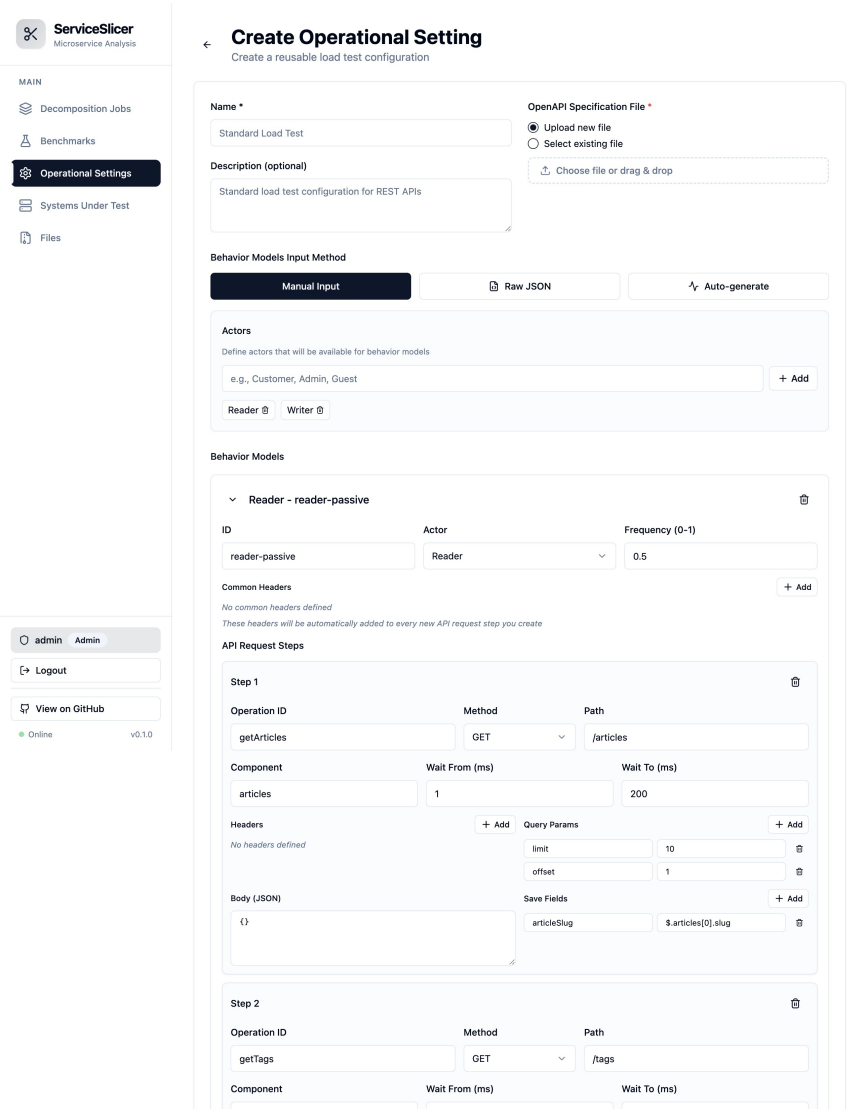

Figure C.2: Operational setting configuration page showing behavior model definition and load level specification

C.3 Benchmark Run Dashboard

The benchmark run dashboard (Figure C.3) provides real-time monitoring of benchmark execution and presents aggregated performance results upon completion. This interface is the central location for observing scalability assessment progress and comparing architectural variants.

During execution, the dashboard displays the current state of each test suite and test case. A test suite groups all load-level tests for a single system under test, while individual test cases represent specific SUT-load level combinations. Status indicators show whether each test is pending, running, completed, or failed, enabling users to monitor progress across the entire benchmark run.

The results section presents performance metrics organized at two levels of granularity. At the system level, users can compare the overall domain metric across architectural variants, providing a high-level assessment of relative scalability. Below the system-level summary, operation-level metrics are displayed in tabular format, organized into two views: aggregated metrics across all load levels for each operation, and detailed per-test-case metrics showing performance at specific concurrency levels. These tables include mean response time, standard deviation, throughput, and pass/fail classification against scalability thresholds.

Interactive charts visualize performance trends across load levels, making it easy to identify scalability bottlenecks and understand how different architectures respond to increasing concurrency. The scalability footprint chart shows the maximum sustainable load for each operation, while response time distribution plots reveal latency characteristics under various conditions.

A comparison view enables side-by-side analysis of the baseline architecture against decomposed alternatives, highlighting operations where the microservices architecture improves or degrades performance relative to the monolith.

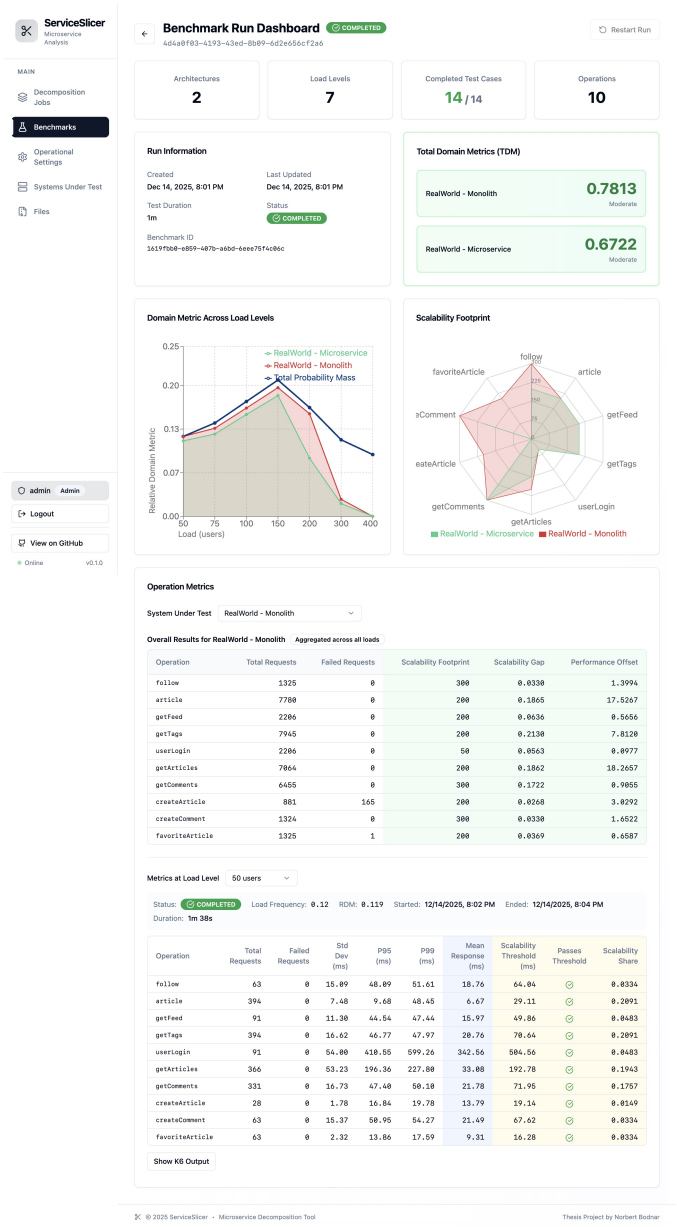

Figure C.3: Benchmark run dashboard displaying test execution progress and aggregated performance metrics across architectural variants

Bibliography

1. DRAGONI, Nicola; GIALLORENZO, Saverio; LLUCH-LAFUENTE, Alberto; MAZZARA, Manuel; MONTESI, Fabrizio; MUSTAFIN, Ruslan; SAFINA, Larisa. Microservices: yesterday, today, and tomorrow. In: 2017.
2. HIGHTOWER, Kelsey. *Microservices Adoption in 2020*. 2020. Available also from: https://www.oreilly.com/radar/microservices-adoption-in-2020/.
3. NEWMAN, Sam. *Building Microservices*. 1st. O'Reilly Media, Inc., 2015. ISBN 1491950358.
4. FOWLER, Martin. *Microservice Trade-Offs* [https://martinfowler.com/articles/microservice-trade-offs.html]. 2015. Accessed 2025-01-15.
5. ABGAZ, Yalemisew; MCCARREN, Andrew; ELGER, Peter; SOLAN, David; LAPUZ, Neil; BIVOL, Marin; JACKSON, Glenn; YILMAZ, Murat; BUCKLEY, Jim; CLARKE, Paul. Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review. *IEEE Transactions on Software Engineering*. 2023, vol. 49, no. 8, pp. 4213–4242. Available from DOI: 10.1109/TSE.2023.3287297.
6. AUER, Florian; LENARDUZZI, Valentina; FELDERER, Michael; TAIBI, Davide. From monolithic systems to Microservices: An assessment framework. *Information and Software Technology*. 2021, vol. 137, p. 106600. ISSN 0950-5849. Available from DOI: https://doi.org/10.1016/j.infsof.2021.106600.
7. TAIBI, Davide; LENARDUZZI, Valentina; PAHL, Claus. Processes, Motivations and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing*. 2017, vol. 4. Available from DOI: 10.1109/MCC.2017.4250931.
8. LOUKIDES, Mike. Technology Trends for 2024 [O'Reilly Radar Report]. 2024. Available also from: https://www.oreilly.com/radar/technology-trends-for-2024/. "61% of enterprises adopted microservices, yet 29% reported returning to monoliths".

9. AVRITZER, Alberto; FERME, Vincenzo; JANES, Andrea; RUSSO, Barbara; HOORN, André van; SCHULZ, Henning; MENASCHÉ, Daniel; RUFINO, Vilc. Scalability Assessment of Microservice Architecture Deployment Configurations: A Domain-based Approach Leveraging Operational Profiles and Load Tests. *Journal of Systems and Software*. 2020, vol. 165, p. 110564. ISSN 0164-1212. Available from DOI: https://doi.org/10.1016/j.jss.2020.110564.

10. KALSKE, Miika; MÄKITALO, Niko; MIKKONEN, Tommi. Challenges When Moving from Monolith to Microservice Architecture. In: 2018, pp. 32–47. ISBN 978-3-319-74432-2. Available from DOI: 10.1007/978-3-319-74433-9_3.

11. KALIA, Anup; XIAO, Jin; KRISHNA, Rahul; VUKOVIC, Maja; BANERJEE, Debasish. Mono2Micro: a practical and effective tool for decomposing monolithic Java applications to microservices. In: 2021, pp. 1214–1224. Available from DOI: 10.1145/3468264.3473915.

12. RICHARDSON, Chris. *Microservices Patterns: With Examples in Java*. Manning Publications, 2018.

13. NYGARD, Michael T. *Release It!: Design and Deploy Production-Ready Software*. 2nd. Pragmatic Bookshelf, 2018. ISBN 978-1680502398.

14. BROOKS, Frederick P. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. 20th Anniversary Edition. Addison-Wesley, 1995. Includes discussion related to Conway's Law.

15. CORTEX. *Monoliths vs. Microservices: Differences, Pros & Cons, and Choosing the Right Architecture*. Cortex, 2021-01. Available also from: https://www.cortex.io/post/monoliths-vs-microservices-whats-the-difference. Accessed: 2025-12-13.

16. GYSEL, Michael; KÖLBENER, Lukas; ZIMMERMANN, Olaf. Service Cutter: A Systematic Approach to Service Decomposition. In: 2016, pp. 185–200. ISBN 978-3-319-44481-9. Available from DOI: 10.1007/978-3-319-44482-6_12.

17. EVANS, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

18. CAMILLI, Matteo; COLARUSSO, Carmine; RUSSO, Barbara; ZIMEO, Eugenio. Actor-Driven Decomposition of Microservices through Multi-level Scalability Assessment. *ACM Trans. Softw. Eng. Methodol.* 2023, vol. 32, no. 5. ISSN 1049-331X. Available from DOI: 10.1145/3583563.

19. HOMBERGS, Tom; STARKE, Gernot. *Get Your Hands Dirty on Clean Architecture: Build 'clean' applications with code examples in Java*. 2023.

20. TRAAG, Vincent A; WALTMAN, Ludo; VAN ECK, Nees Jan. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific Reports*. 2019, vol. 9, no. 1, p. 5233. Available from DOI: 10.1038/s41598-019-41695-z.

21. BLONDEL, Vincent D; GUILLAUME, Jean-Loup; LAMBIOTTE, Renaud; LEFEBVRE, Etienne. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*. 2008, vol. 2008, no. 10, P10008. Available from DOI: 10.1088/1742-5468/2008/10/P10008.