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.

Bc. Norbert Bodnár

Advisor: doc. Bruno Rossi, PhD

Acknowledgements

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

Software systems have evolved significantly over the past decades, with many organizations initially building large monolithic applications that served their business needs. However, as these systems grew in complexity and size, they began to present numerous challenges in terms of maintenance, scalability, and deployment [1]. This has led to a significant shift towards microservices architecture [2], which promises better scalability, maintainability, and deployment 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. Moreover, 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].

Recent industry observations highlight the uncertainty surrounding architectural transitions. O'Reilly's Technology Trends for 2024 [8] reports that although 61% of surveyed enterprises have adopted microservices, 29% have encountered difficulties severe enough to initiate a return to monolithic architectures. This demonstrates that architectural transitions carry substantial risks: the resulting system may become more complex to operate, exhibit worse performance characteristics, or fail to deliver the 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 appears logically sound, its runtime behavior may diverge significantly once services are deployed and subjected to operational traffic. Communication patterns, serialization costs, database contention, and service 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 the growing interest in decomposing monolithic systems into microservices [5] [6] [1], organizations often lack systematic and reliable means to evaluate the quality and viability of proposed decompositions before committing to 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, practical tooling that combines static decomposition analysis with automated, reproducible benchmarking of multiple architectural variants appears to be scarce [5]. Current practices frequently depend on intuition, ad-hoc experiments, or manual prototyping, which makes it challenging to compare alternatives consistently or

detect performance regressions introduced by decomposition choices. 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.

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 as follows: TODO

Chapter 2 reviews 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 Service Slicer platform, detailing its components, data flows, and integration with external tools for static analysis and benchmarking.

Chapter 4 describes the implementation of Service Slicer, including key algorithms, data models, and workflow orchestration mechanisms.

Chapter 5 evaluates Service Slicer through case studies on real-world monolithic applications, assessing its effectiveness in generating decomposition candidates and measuring scalability under load.

Chapter 6 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, surveys existing solution approaches, and identifies gaps that motivate the design of ServiceSlicer. We begin by exploring the multifaceted nature of decomposition challenges before reviewing how the research community has attempted to address them.

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. 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 [13] 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 [14]. 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 [15] [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.

2.2 Existing Solution Approaches

Despite these challenges, researchers have developed multiple approaches for identifying service boundaries. 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. The following subsections summarize the major categories identified in the M2MDF systematic review [5].

2.2.1 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 [16]), 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 [17]), 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.

2.2.2 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 documenta-

tion 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.

2.2.3 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.

2.2.4 Version History Mining

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 approach addresses specific aspects of the decomposition problem, but none provides end-to-end support for the complete migration lifecycle. Recognizing this, recent research has attempted to synthesize these techniques into unified frameworks.

2.3 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 review 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.

2.3.1 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 emu-

lated 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.

2.4 Evaluating Decomposition Quality Through Scalability Assessment

The lack of deployment-based validation identified in M2MDF stems partly from the absence of systematic methodologies for comparing architectural variants under realistic load. Multi-Level Scalability Assessment [9] provides such a methodology, enabling engineers to evaluate whether a proposed decomposition delivers measurable improvements over the original monolith.

Rather than treating scalability as a single aggregate property, the methodology analyzes system behavior at three levels: **system-level** (overall throughput and response time), **component-level** (service or module performance), and **operation-level** (individual API endpoint behavior). This multi-level perspective enables engineers to identify not only whether a system scales, but which specific operations or components limit scalability.

The methodology operates by defining an **operational profile** that captures realistic usage patterns through behavior models (sequences of API operations representing user workflows), behavior mix (relative frequencies of different workflows), and load distribution (concurrent user levels and their probabilities). Load tests are executed across multiple architectural variants at various concurrency levels. A

baseline architecture establishes performance thresholds at minimal load, against which all architectures—including the baseline itself at higher loads—are evaluated. The framework computes metrics including domain metric (aggregate scalability satisfaction), scalability footprint (maximum sustainable load per operation), scalability gap (performance degradation at specific load levels), and performance offset (severity of threshold violations).

By analyzing scalability at multiple abstraction levels, this methodology enables architects to identify not just whether a system scales, but which specific operations or components limit scalability. However, 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 Service Slicer — System Architecture & Design

This chapter presents the architecture and design of the Service Slicer platform, building on the decomposition challenges and methodologies discussed previously. It describes the system's structure, key components, and data flows, explaining how Service Slicer integrates static analysis and empirical benchmarking to support monolith decomposition. The following sections provide a detailed overview of each architectural element and their roles in enabling reproducible evaluation of candidate microservice boundaries.

3.1 Overview

Service Slicer 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, Service Slicer 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.

3.2 Workflows

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

The **static analysis workflow** begins with the upload of a monolithic application artefact. This artefact is stored in object storage, while metadata and job state are persisted in the relational database. The workflow proceeds asynchronously: dependency extraction pro-

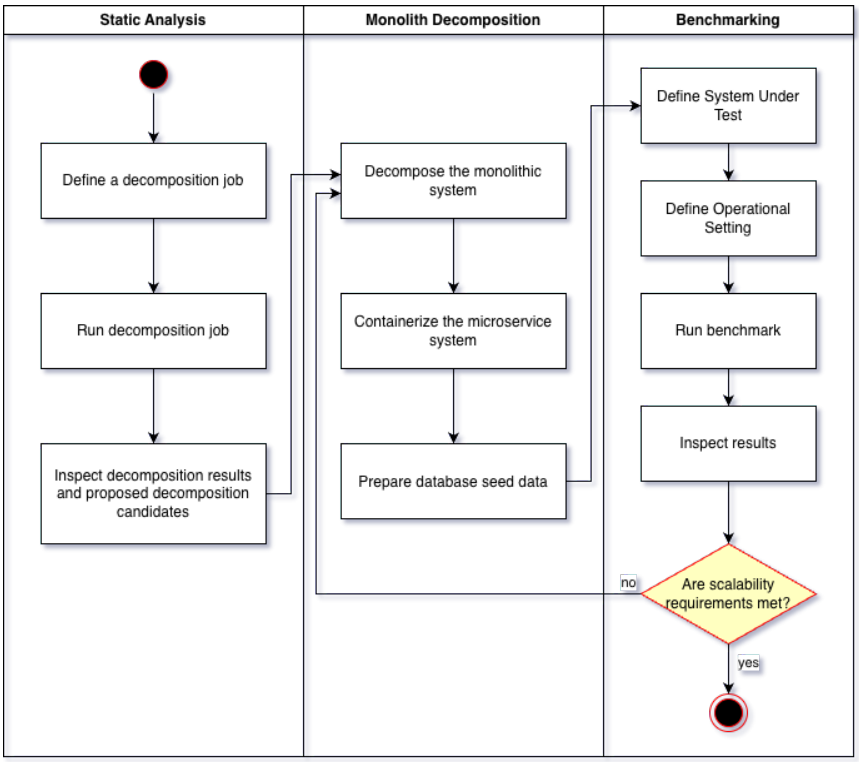

Figure 3.1: Diagram of the Workflows in Service Slicer

duces a graph representation stored in Neo4j, community detection algorithms identify candidate service boundaries by clustering the graph, and optional AI-based refinement applies domain-driven or actor-driven decomposition strategies. Throughout this process, intermediate and final results—including dependency graphs, clustering outcomes, and calculated metrics—are written to persistent storage. Users retrieve consolidated decomposition results through read queries that aggregate data from both the graph database and the relational database.

The **benchmarking workflow** evaluates the runtime behavior of deployable system architectures. Users supply configuration artefacts—Docker Compose files, OpenAPI specifications, and database seeds—which are stored in object storage, with references maintained in the relational database. The workflow operates in three phases: validation confirms that each system under test conforms to the opera-

tional 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. Metrics collected during load testing are first captured by Prometheus from the deployed systems and are subsequently aggregated and persisted in the relational database. Read-only query endpoints then expose these stored results, enabling access to performance summaries, comparative visualisations, and computed scalability metrics.

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 Chapter 4.

3.3 System Assumptions and Operational Constraints

The design of Service Slicer 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 Service Slicer to control startup, resource allocation, health checks, and teardown in a con-

sistent 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 Service Slicer 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

Service Slicer 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.

Service Slicer integrates several cooperating components and external tools, as depicted in Figure 3.2. The system follows a client-server ar-

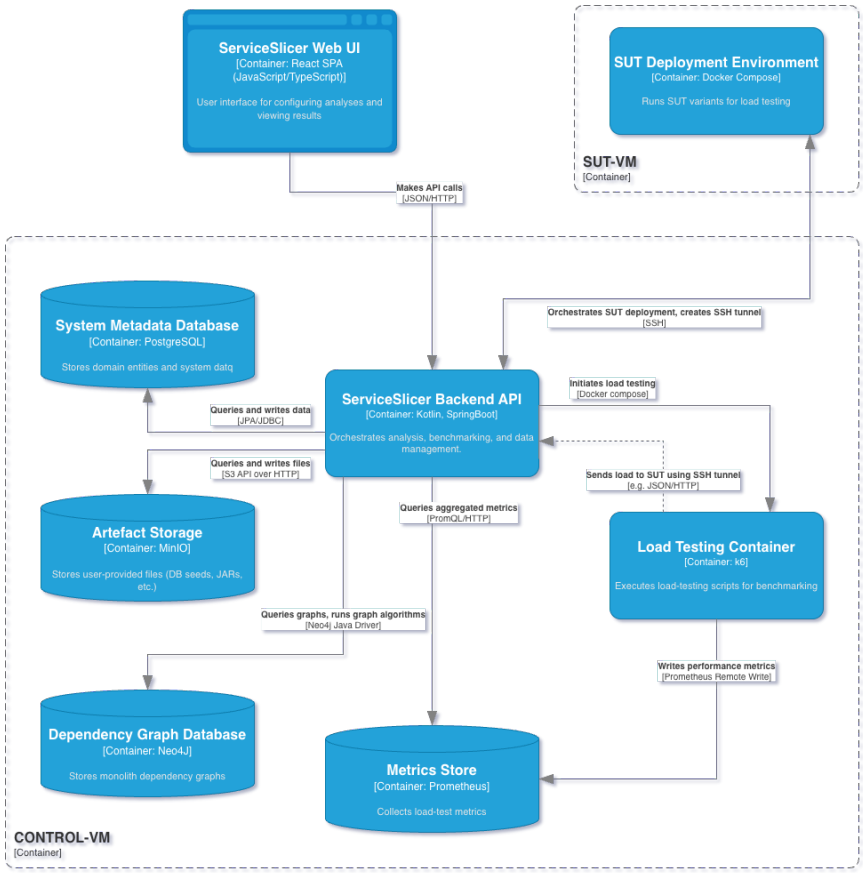

Figure 3.2: High-Level Architecture of Service Slicer

chitecture: 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[1] for domain entities and workflow state, Neo4j[2] for dependency graphs, MinIO[3] for large binary artefacts, and Prometheus[4] for performance metrics. During benchmarking, the backend automatically deploys Systems Under Test using

1. `https://www.postgresql.org/`
2. `https://neo4j.com/`
3. `https://min.io/`
4. `https://prometheus.io/`

Docker Compose, executes load tests with k6[5], 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

This section describes the individual components that constitute the Service Slicer platform, explaining their responsibilities, internal technologies, and design rationale.

3.5.1 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. It is used in ServiceSlicer to offer a responsive and intuitive environment for navigating complex decomposition results and controlling long-running workflows.

3.5.2 Backend

The backend is a Kotlin/Spring Boot service that orchestrates all workflows in ServiceSlicer. It implements the domain logic for decomposition jobs, benchmark execution, operational profile management, and scalability analysis. Using a modular, hexagonal architecture (Figure 3.3), it coordinates interactions with databases, storage systems, and external tools. It is used as the central engine that executes static analysis, manages load-test automation, aggregates metrics, and exposes the REST API consumed by the frontend.

3.5.3 Neo4j — Graph Database

Neo4j stores the dependency graph extracted from the monolithic application, representing classes and their relationships. It executes

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

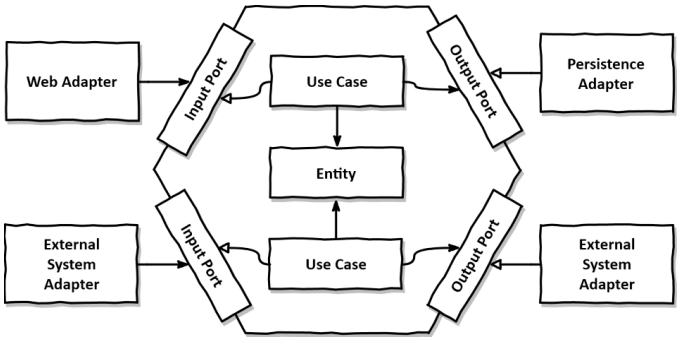

Figure 3.3: Hexagonal Architecture, reproduced from [18]

community detection algorithms to identify clusters that may represent candidate microservices. It is used in ServiceSlicer because graph-native storage and analytics enable efficient traversal, pattern querying, and algorithmic decomposition—capabilities that would be cumbersome in a relational model.

3.5.4 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 is used in ServiceSlicer as the authoritative store for workflow state and as the source of truth for all user-visible results.

3.5.5 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. It is used in ServiceSlicer to efficiently store and retrieve artefacts that are too large or too unstructured for relational storage.

3.5.6 K6 + Prometheus

K6 is the load-testing engine that executes behavior-model-driven test scenarios during benchmarking. Prometheus collects fine-grained time-series metrics produced by K6, such as response times, percentiles, and error rates. Together, they are used in ServiceSlicer to measure the runtime behavior of each architectural candidate under controlled load and to compute the multi-level scalability indicators.

3.5.7 Docker Compose

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. It is used in ServiceSlicer to guarantee consistent testing conditions, simplify environment provisioning, and enable automated end-to-end benchmark execution.

4 Static Code Analysis Workflow

The static code analysis workflow extracts structural dependencies from a monolithic Java application and identifies candidate microservice boundaries using graph-based clustering algorithms. This process provides architects with data-driven insights into how the system's components interact, enabling informed decisions about service partitioning.

Figure 4.1: Static Analysis Sequence Diagram

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 (Figure 4.2) 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 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.

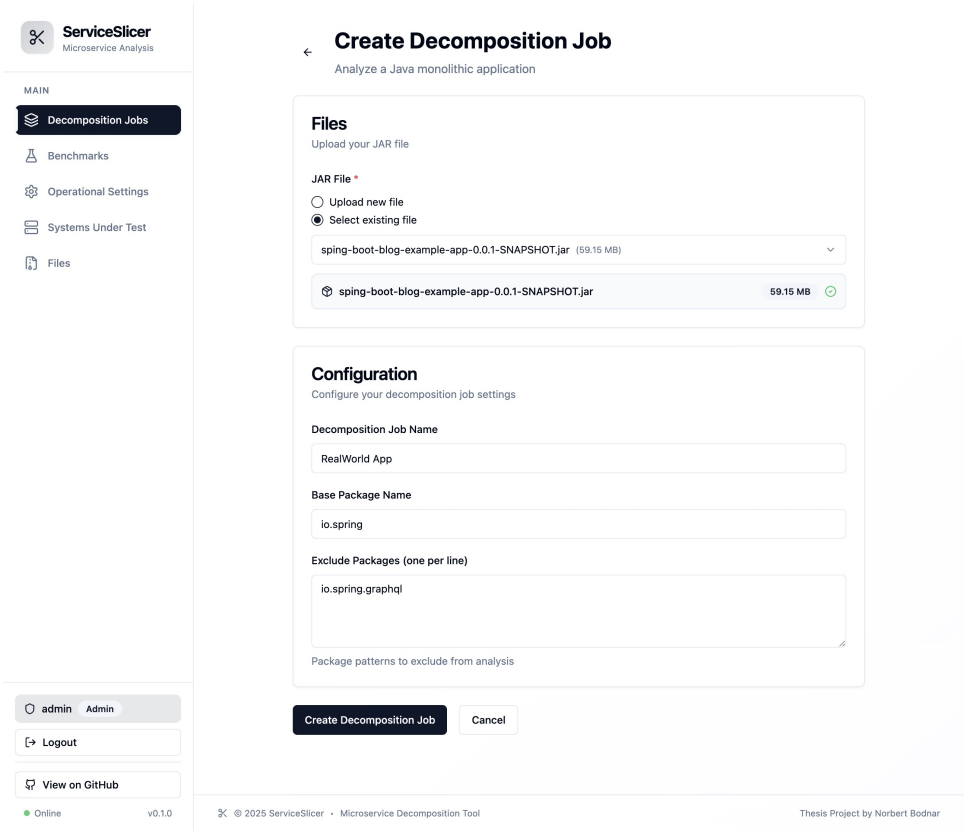

Figure 4.2: Create Decomposition Job page

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” relation-

ship 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**: 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**: 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.

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 4.3. Nodes in the graph represent classes and are colored according to their assigned service in the selected decomposition can-

didate, 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.

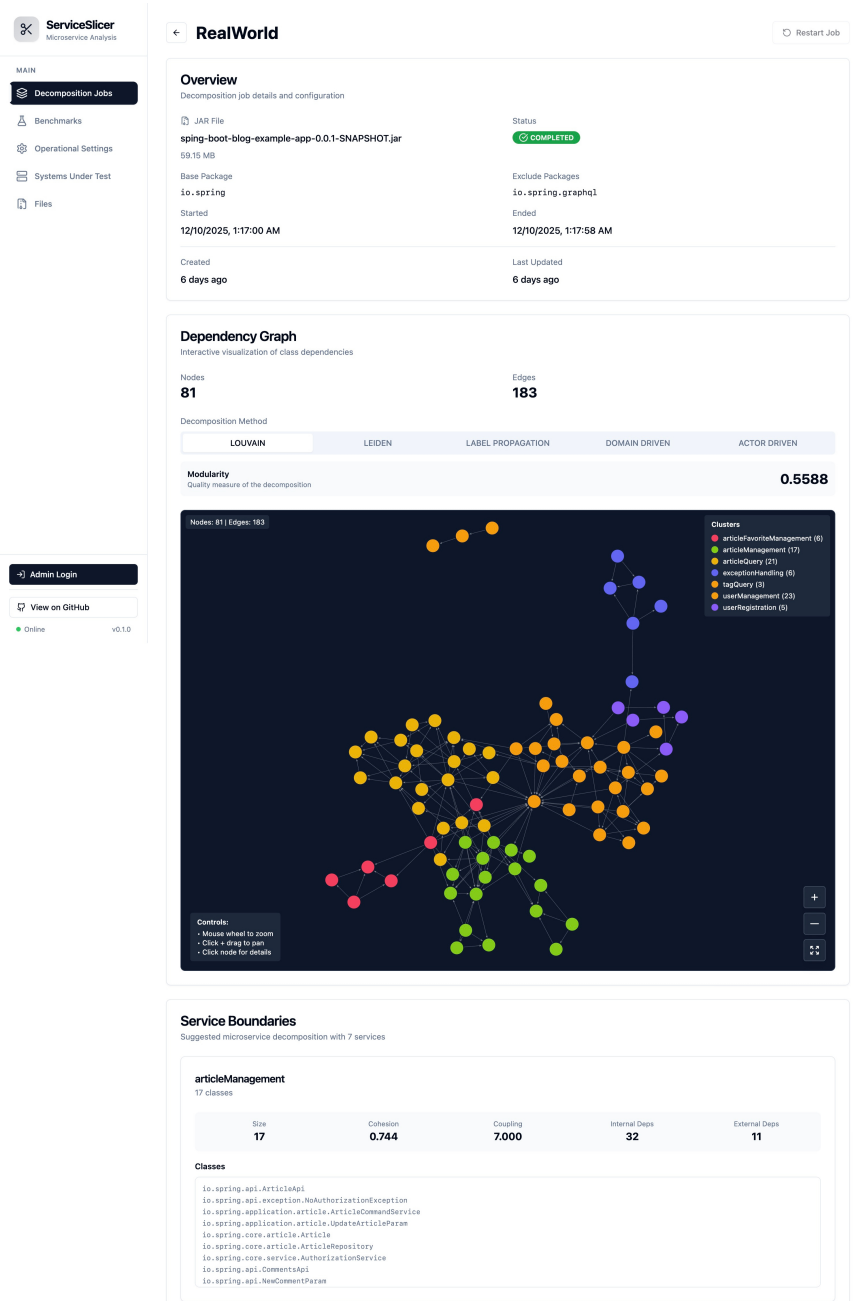

Figure 4.3: Decomposition job details page visualizing candidate microservice boundaries with color-coded clusters

5 Performance Benchmarking Workflow

The automated performance benchmarking workflow evaluates the scalability characteristics of different architectural variants under realistic load conditions. By executing controlled load tests and collecting detailed performance metrics, this workflow provides empirical data to inform architectural decisions based on observed system behavior rather than theoretical assumptions or intuition.

5.1 Methodological Foundation

The benchmarking methodology implemented in this chapter is based on the Multi-Level Scalability Assessment framework introduced by Avritzer et al. [9]. This framework provides a systematic approach for evaluating system scalability at multiple levels of granularity under realistic operational profiles. The framework defines the mathematical formulations for scalability thresholds, domain metrics, share metrics, and weighted aggregation procedures that enable comparative assessment of architectural variants.

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 actor-driven behavior models; (3) systematic collection and aggregation of performance metrics; and (4) integration with the static decomposition analysis workflow described in the previous chapter. The formulas, metrics, and multi-level assessment procedures presented in this chapter are drawn directly from the referenced methodology and are implemented as specified in the original work.

5.2 Workflow Overview

The benchmarking workflow is designed around the principle of comparative evaluation: it measures the performance of multiple architec-

tural 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 (Figure 5.1). 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 defines the workload model applied during benchmarking (Figure 5.2). It formalizes expected usage patterns by specifying which actors interact with the system, what sequences of operations they perform, and how frequently these patterns occur under realistic conditions. The specification is grounded in an OpenAPI contract that defines all exposed operations and serves as both a validation artifact and the foundation for workload construction. The operational setting comprises two components: behavior models and an operational profile.

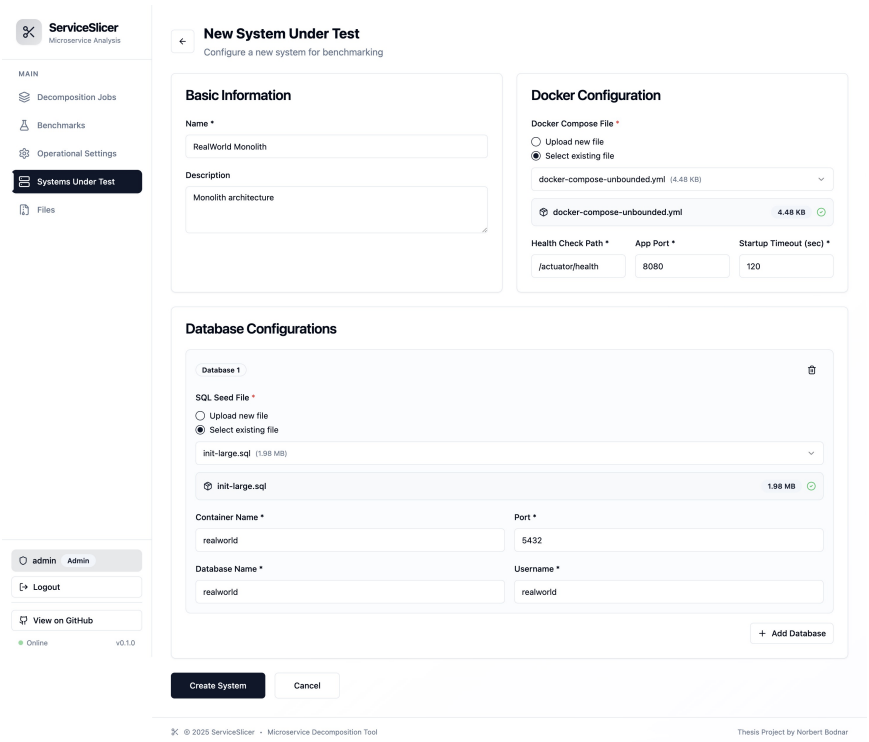

Figure 5.1: System under test configuration interface

Behavior Models

Behavior models formalize user interaction patterns as sequences of API operations that actors execute to accomplish specific goals. Each model captures the temporal ordering of operations, data dependencies between requests, and inter-operation delays.

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, though refinement based on observed patterns is recommended for production use.

Each behavior model specifies the actor type (e.g., customer, administrator), relative frequency of occurrence, and the complete operation sequence. Operations include HTTP methods, endpoint paths, parameters, headers, and request bodies. Data dependencies are captured through variable bindings, allowing values from one response to be injected into subsequent requests—enabling stateful workflows like authentication followed by resource manipulation. Stochastic think times between operations model realistic user pauses.

Operational Profile

The operational profile defines the load distribution applied during benchmarking. It specifies discrete load levels—each representing a target number of concurrent virtual users—along with the probability that each load level occurs under realistic production conditions. This probabilistic characterization enables weighted aggregation of performance metrics, ensuring that results emphasize the most frequently encountered load levels rather than treating all scenarios uniformly.

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 (Figure 5.3). 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 Γ_j^0 for each operation j:

$$\Gamma_j^0 = \mu_j^0 + 3\sigma_j^0 \tag{5.1}$$

where μ_j^0 is the mean response time and σ_j^0 is the standard deviation observed during the baseline's minimum load test. 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 comprising deployment, workload execution, metrics collection, and teardown phases:

System Deployment and Initialization The first step in each test case deploys the designated SUT and verifies its operational readi-

ness. For remote execution scenarios, all required deployment artefacts—Docker Compose descriptors, database seed scripts, and configuration files—are transferred to the target host. The deployment process instantiates all application and infrastructure containers as specified in the Docker Compose file, allocating the necessary compute, memory, and network resources.

Once containers are running, the health-check endpoint is polled repeatedly until the system signals readiness or the configured startup timeout is reached. This verification step ensures that the application has completed initialization, established database connections, and is prepared to accept requests before load testing commences, preventing measurement artifacts caused by initialization overhead.

If database seeding is configured, initialization scripts are executed against the designated database instances to populate them with representative application state. Seeding occurs after successful health checks to ensure that database schemas have been initialized. This mechanism guarantees that each test run begins from a consistent and reproducible data state, eliminating variability introduced by differences in data volume or distribution across test cases.

Workload Generation and Execution Following successful deployment, the load testing framework executes the workload defined in the operational setting. The workload generator is configured with the operational setting, the architecture identifier, the target concurrency level, the test duration, and a ramp-up period during which transient startup effects are excluded from measurement.

The load test simulates realistic user interactions by spawning multiple concurrent virtual users, each executing behavior models repeatedly for the duration of the test. Virtual users probabilistically select behaviors according to the usage profile weights specified in the operational setting, ensuring that the aggregate workload reflects realistic usage distributions. Each virtual user executes the selected behavior as a sequence of API operations, following the temporal ordering and data dependencies defined in the behavior model.

For each operation, the load generator constructs an HTTP request by substituting path and query parameters with values extracted from prior responses or predefined constants, and including the specified

headers and request body. The request is dispatched to the SUT, and response timing information is recorded. All requests are tagged with metadata identifying the operation, actor, behavior, architecture variant, and load level, enabling subsequent fine-grained metric aggregation and multi-level scalability analysis.

Between consecutive operations, virtual users pause for a stochastic think time sampled from the configured inter-operation delay interval. This introduces realistic temporal patterns that reflect user reading, decision-making, or external processing activities, preventing the artificially synchronized request bursts that would result from continuous, uninterrupted operation execution. Once a behavior sequence completes, the virtual user repeats the workflow, continuing until the configured test duration elapses. This continuous execution model ensures sustained load generation and enables observation of steady-state system behavior under realistic operational conditions.

Performance Metric Computation Upon test completion, the load testing framework computes aggregated performance statistics from the collected response time measurements. Data recorded during the ramp-up period is excluded to ensure that only steady-state performance is captured. For each operation j, the following metrics are computed:

- **Mean response time** (μ_j): The arithmetic mean of all steady-state response time samples for operation j.
- **Standard deviation** (σ_j): A measure of response time variability, computed as the square root of the variance of the collected samples.
- **Invocation frequency** (ν_j): The relative frequency of operation j within the workload, representing its contribution to the overall operational profile.
- **Scalability threshold**: For the baseline architecture at the lowest load level, a threshold is computed for each operation as a function of its mean response time and standard deviation. These thresholds define acceptable performance bounds and are reused for all subsequent evaluations.

- **Operation pass/fail status**: For each architecture and load level, an operation is classified as passing or failing depending on whether its mean response time exceeds the corresponding scalability threshold.
- **Scalability share**: The contribution of operation j to overall scalability at a given load, computed as the product of its invocation frequency and pass/fail status.
- **Scalability footprint**: An operation-level metric that captures the maximum load level at which each operation remains scalable, providing insight into how individual operations degrade under increasing load.
- **Scalability gap**: A load-level metric that quantifies the loss of scalability due to failing operations by comparing the achieved scalability share with the ideal case in which all operations pass.
- **Performance offset**: For failing operations, the amount by which the observed mean response time exceeds the scalability threshold, indicating the severity of the scalability violation.
- **Relative domain metric**: An architecture-level metric that expresses how well a system satisfies scalability requirements relative to the baseline architecture. It is computed by normalizing the architecture's aggregated scalability shares against those of the baseline.
- **Total domain metric**: An aggregate metric obtained by combining the relative domain metrics across all evaluated architectures, weighted by the operational profile, to provide a holistic view of scalability within the benchmark.

Formal definitions and mathematical details are provided in the original Multi-Level Scalability Assessment methodology [9].

Result Persistence and System Teardown Once all metrics have been collected, the test case is marked as completed and its results are persisted to the relational database. Each test case record stores a comprehensive set of performance data.

Following result persistence, the SUT is systematically torn down: all running containers are stopped and their associated storage volumes are removed. This cleanup ensures a pristine execution environment for subsequent test cases, preventing state leakage or residual data from contaminating later measurements. The teardown procedure is guaranteed to execute even in the event of test failures, maintaining environmental hygiene throughout the benchmark run.

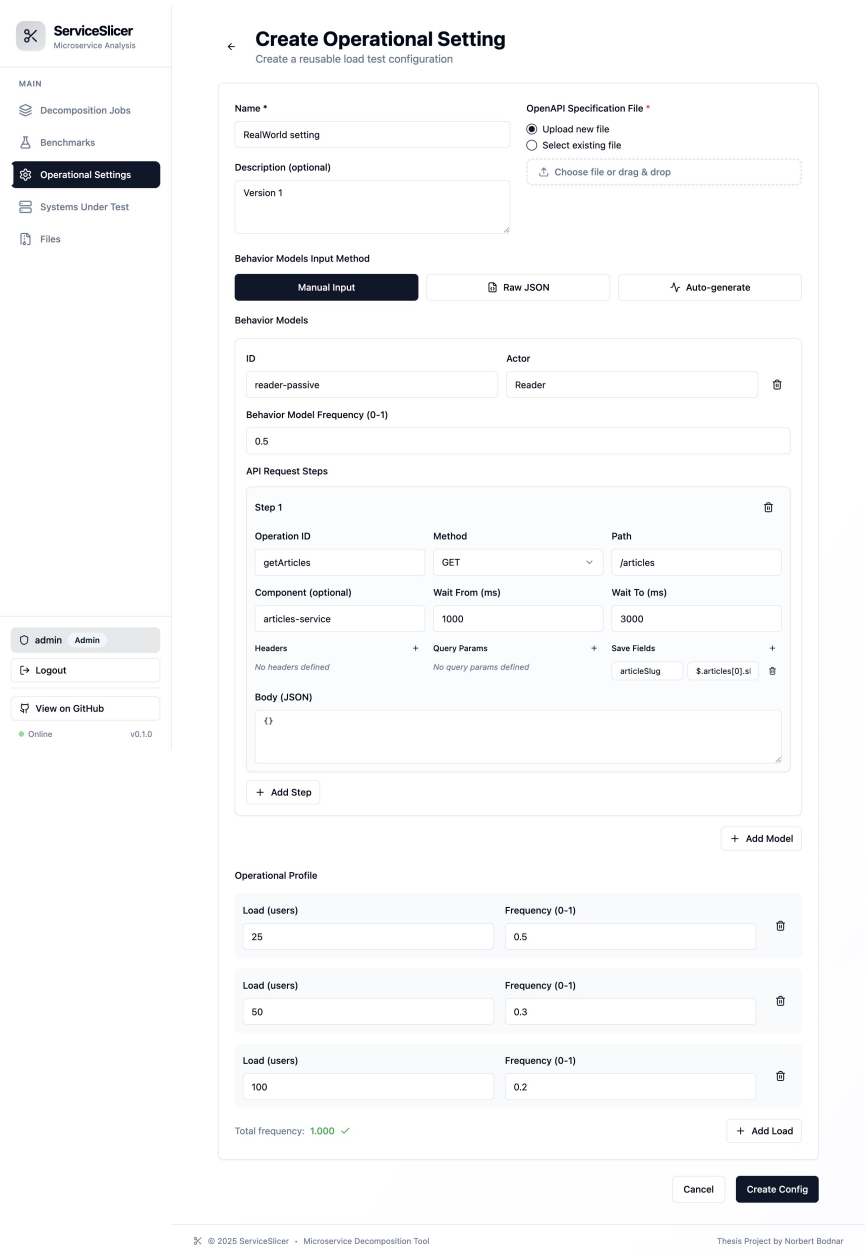

Figure 5.2: Operational setting configuration interface

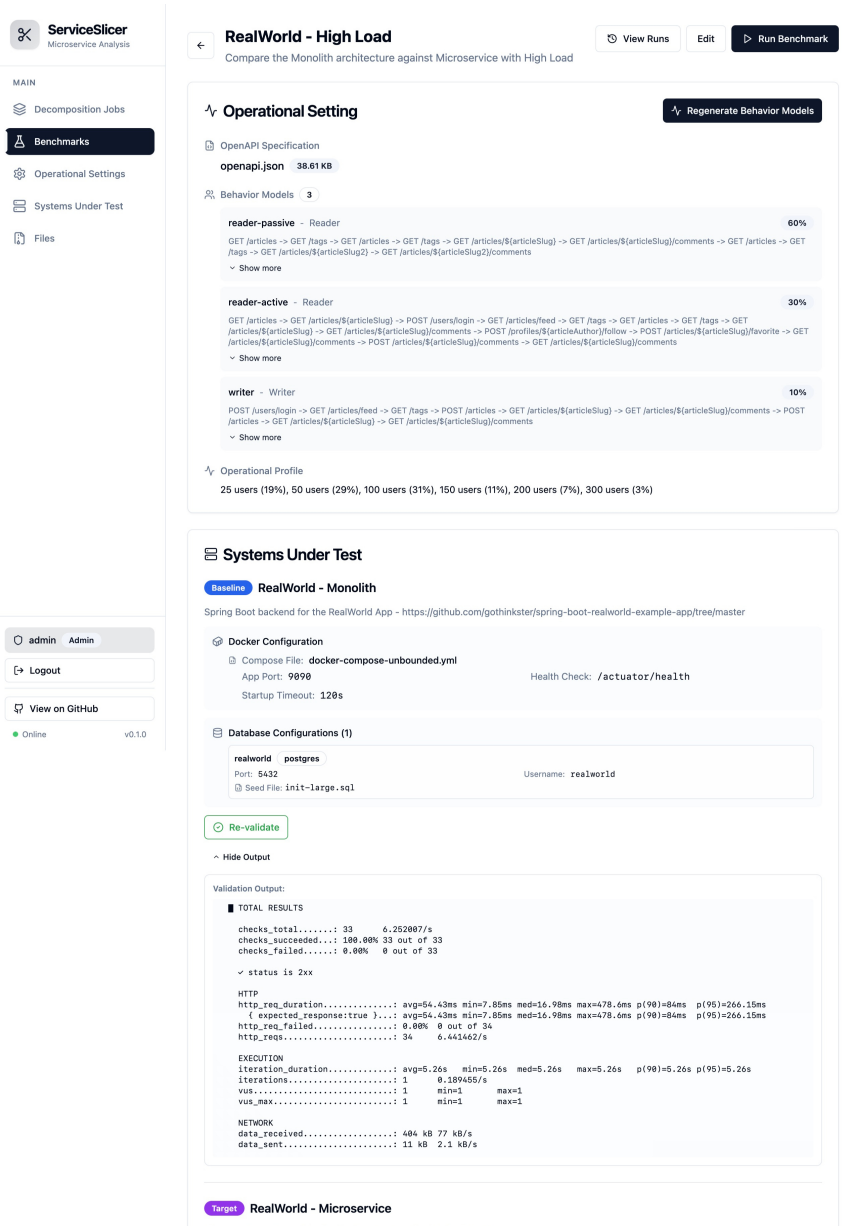

Figure 5.3: Benchmark interface showing validation status and diagnostic outputs

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 of different scales and 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 real-world complexity, including domain-specific business logic, operational constraints, and the trade-offs that emerge when evaluating architectural alternatives for systems with established user bases and performance requirements.

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 Conduit, walking through the complete workflow from dependency analysis to perfor-

mance comparison. Section 6.2 demonstrates the tool's application to Technika, highlighting how it addresses industrial system complexity. Section 6.3 synthesizes observations across both examples, discussing patterns, discovered limitations, and practical implications for practitioners considering similar decomposition efforts. Finally, Section 6.4 reflects on the scope and limitations of these demonstrations and the boundaries within which the presented findings should be interpreted.

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.

Figure 6.1: Leiden algorithm decomposition of the RealWorld Conduit application

In contrast, the Label Propagation algorithm identified only two clusters, one of which contained the majority of classes. This result reflects the existence of a densely connected core within the application, with only a small subset of classes forming a loosely coupled peripheral group. The divergence between modularity-based and diffusion-based clustering outcomes highlights that, while the system exhibits strong overall cohesion, finer-grained structural partitions can still be extracted when optimizing for modularity. Figure 6.1 illustrates the decomposition produced by the Leiden algorithm, which is representative of the modularity-based results and serves as a basis for subsequent analysis.

6.1.3 Generated Decomposition Candidates and Implementation

The algorithmic clustering results and AI-assisted decomposition suggestions obtained during the static analysis phase served as guidance for designing a microservices architecture. The Leiden algorithm's output, which achieved the highest modularity score and produced structurally coherent clusters, was interpreted alongside domain knowledge of the blogging platform to derive a decomposition that is both structurally justified and practically deployable.

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, user interactions through comments, and user management represent distinct areas of functionality with well-defined responsibilities and data ownership.

The services are deployed behind an API gateway that exposes the same REST API as the original monolithic application, ensuring functional equivalence and enabling fair performance comparison. Each microservice manages its own database, enforcing data ownership and eliminating direct cross-service database access.

The resulting microservice-based application was containerized using Docker Compose to support automated deployment during benchmarking. In preparation for performance evaluation, representative seed data was generated and loaded into the respective databases to ensure that load tests were executed against a realistic dataset rather than an empty or trivial state.

6.1.4 Definition of the Operational Setting

The operational setting for the RealWorld Conduit application was defined to approximate realistic system usage based on the existing REST API and observed interaction patterns. To this end, the deployed demo version of the application was used to simulate representative user sessions, during which network requests issued by the frontend were recorded and analyzed.

Based on this analysis, two primary actor types were identified: a *reader* and a *writer*. These actors were further refined into three distinct behavior models. The *reader-passive* behavior represents passive

consumption and consists of reading an article and retrieving its associated metadata. The *reader-active* behavior extends this flow by including interactive actions such as posting comments and following authors. The *writer* behavior captures content creation activities, including author authentication and article publication. Each behavior model was defined as an ordered sequence of API operations corresponding to the recorded request traces.

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

The extracted request sequences served as the basis for constructing the operational profile, ensuring that the generated load reflects realistic mixes of read and write interactions. To determine a feasible workload intensity given the available computational resources, a series of preliminary load tests were conducted. These diagnostic runs were used to identify a load level at which the system remained responsive without exhibiting excessive error rates or resource exhaustion.

Based on the identified sustainable load, the final operational profile was defined using a normal distribution centered around this value. This approach models typical fluctuations in user activity while avoiding unrealistically extreme load conditions, and it provides a stable foundation for subsequent scalability assessment.

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 → 06
Writer	writer	0.2	o5 → o10 → o2 → o9 → o3 → o4

6.1.5 Benchmark Execution and Results

The benchmarking workflow was executed following the procedure outlined in Chapter 5. The test cases were configured to run for a 30s warm-up period followed by a 5-minute measurement phase at each load level. Benchmarks compared each candidate architecture against the original monolithic baseline, executing load tests at all specified load levels and collecting performance metrics.

[TODO: Present performance measurements and scalability assessment outcomes]

6.1.6 Observations and Insights

[TODO: Discuss what the results reveal about the decomposition and the tool's utility]

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.

6.2.3 Generated Decomposition Candidates and Implementation

The algorithmic clustering results served as initial guidance for identifying service boundaries, but the final decomposition strategy was developed in coordination with a domain expert who understood the system's operational context and business requirements. This collaborative approach enabled the integration of structural insights from static analysis with domain knowledge about functional cohesion, operational workflows, and data ownership patterns.

Through this process, a decomposition strategy emerged that aligned with the two primary actor types interacting with the system. Rather than pursuing fine-grained decomposition based solely on structural clustering, the decision was made to extract two coarser-grained services that corresponded to distinct operational responsibilities:

- **Admin Service**: Responsible for managing master data synchronization with external systems, handling devices, employees, and projects. This service encapsulates the system's administrative interface used primarily by automated external processes for bulk data updates and periodic synchronization operations.
- **Worker Service**: Responsible for operational workflows performed by field workers, including transaction management (device usage tracking, fuel consumption, mileage recording) and work log entries (time tracking, project assignment). This service supports the high-frequency, interactive operations that constitute the majority of day-to-day system usage.

This actor-aligned decomposition strategy reflects a pragmatic balance between architectural clarity and operational feasibility. By organizing services around distinct user personas rather than pursuing maximal structural decomposition, the design maintains clear service boundaries while limiting the operational complexity that would arise from managing numerous fine-grained microservices in a production environment.

Based on this strategy, a microservices implementation was developed consisting of the Admin Service and Worker Service as described above. Each microservice was containerized using Docker Compose,

with deployment configurations specifying resource allocations, network topologies, and database initialization procedures. To ensure realistic testing conditions, the benchmark environment was initialized using a database dump extracted from the production server, providing representative data volume and distribution patterns that reflect actual system usage.

This approach enabled comparative assessment between the original monolithic architecture and the proposed microservices decomposition under conditions closely approximating production workloads, offering insights into how the architectural transition might affect system performance in operational deployment.

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 External System Actor and Common Worker Actor.

Table 6.3: 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

These behavior models formalize the workflows performed by each actor type, including the sequential ordering of API operations,

parameter dependencies between requests, and realistic think times reflecting human interaction patterns or automated processing delays. The behavior mix was weighted according to observed usage frequencies, with worker operations constituting the majority of system traffic (90%) and external system synchronization occurring at lower frequency (10%) but with distinct performance characteristics.

Table 6.4: Behavior models and operation sequences used in the Technika load testing

Actor	**Behavior model**	**Behavior mix**	**Steps (operation shortcuts)**
External System	sync-data-write	0.05	o1 → o2 → o2 → o2 → o3 → o4 → o4 → o4 → o5 → o6 → o6 → o6
External System	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

Load levels were specified to reflect realistic operational conditions, ranging from 0 to 300 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 30s warm-up period followed by a 5-minute measurement phase at each load level. Performance metrics were collected and aggregated according to the multi-level scalability assessment methodology.

[TODO: Present performance comparison between monolith and decomposed architecture]

6.2.6 Limitations Encountered

- modifying existing operational settings will break the already finished or running benchmarks. Need to implement better separation of configurations. - maybe we could use better graph algorithms in the future with more configurable options

[TODO: Discuss effort required, configuration challenges, and tool applicability. Address trade-offs between decomposition granularity and performance, operational complexity considerations, and guidance on selecting decomposition strategies.] [TODO: Discuss issues discovered during demonstrations, such as implementation quality of decomposed variants, database schema decisions, and workload representativeness.]

6.2.7 Representativeness of Demonstrated Systems

[TODO: Discuss what these examples do and do not represent. Address limitations related to system selection, domains, and scale.]

6.2.8 Methodological Constraints

[TODO: Discuss limitations of the demonstration approach compared to formal evaluation. Address appropriateness of metrics, accuracy of operational profiles, and validity of scalability thresholds as performance indicators.]

6.3 Summary

[TODO: Synthesize key findings from both demonstrations, emphasizing general insights and patterns that emerged across different systems and domains.]

7 Conclusions & Future Directions

This thesis has presented ServiceSlicer, a platform for supporting monolith decomposition through integrated static analysis and automated performance benchmarking. By combining dependency extraction, graph-based clustering, AI-assisted semantic decomposition, and multi-level scalability assessment within a unified workflow, ServiceSlicer enables architects to explore decomposition strategies and validate their viability under realistic operational conditions.

The platform 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 work makes three primary contributions:

Integrated Decomposition Support Platform. ServiceSlicer implements multiple phases of the M2MDF framework [5], including bytecode-level dependency extraction, graph-based clustering (Louvain, Leiden, Label Propagation), AI-assisted semantic analysis, structural quality metrics, and automated deployment-based validation. The hexagonal architecture integrates Neo4j, PostgreSQL, MinIO, Prometheus, and k6 into a cohesive workflow accessible through a browser-based interface.

Automated Multi-Level Scalability Assessment. The thesis provides an automated implementation of the Multi-Level Scalability Assessment methodology [9], fully automating deployment orchestration, behavior-model-driven workload execution, and computation of operation-level, component-level, and system-level scalability indicators. This reduces evaluation time from days to hours and enables reproducible performance comparison across architectural variants.

Demonstration Through Real Systems. The platform's applicability is demonstrated through RealWorld Conduit (a blogging platform)

and Technika (an industrial fleet management system), validating that ServiceSlicer can process real applications, generate semantically coherent decomposition candidates, and execute automated benchmarks against containerized deployments.

7.2 Key Findings

Static analysis provides guidance, not prescriptions. Clustering algorithms produce structurally coherent partitions, but these require interpretation through domain knowledge to yield practically deployable microservices. The most effective decompositions combined algorithmic results with domain expertise.

Actor-aligned decomposition balances clarity and feasibility. Organizing services around distinct actor types rather than pursuing maximal structural granularity can produce architectures that are both clear and operationally manageable, particularly for smaller systems or teams with limited operational maturity.

Operational profiles require domain knowledge. Constructing realistic behavior models and load distributions requires substantial domain expertise. While AI-assisted synthesis can generate plausible sequences, validation against observed traffic patterns significantly improves assessment quality.

Deployment-based validation reveals non-obvious bottlenecks. Automated benchmarking uncovered performance characteristics unpredictable from static analysis alone—including network latency, serialization overhead, database contention, and cascading failures under load—reinforcing the importance of empirical evaluation.

7.3 Limitations

JVM-only static analysis. Dependency extraction operates exclusively on JVM bytecode, limiting applicability to Java, Kotlin, and Scala. Extending support to other language ecosystems would require implementing language-specific parsers.

Docker Compose deployment model. Benchmarking assumes systems can be deployed via Docker Compose, excluding Kubernetes,

serverless platforms, or geographically distributed deployments that introduce complexities not captured in single-host scenarios.

Manual microservices implementation. ServiceSlicer generates decomposition candidates but does not automatically extract code to produce functioning microservices, requiring significant manual implementation effort.

Limited scope of demonstrations. The two demonstrated systems are relatively small (fewer than 100 classes), use relational databases, and do not exhibit the extreme coupling or legacy complexity typical of large enterprise monoliths.

Absence of formal evaluation. This thesis presents demonstrations rather than controlled empirical studies, validating feasibility without establishing quantitative claims about decomposition quality or comparative effectiveness.

7.4 Future Directions

Multi-language static analysis. Extend dependency extraction to support multiple programming languages through integration with language-agnostic tools or language-specific parsers, enabling analysis of polyglot codebases.

Scalable graph processing. Implement distributed graph processing, incremental clustering algorithms, or graph sampling techniques to enable analysis of systems comprising tens of thousands of classes.

Automated code extraction. Develop tooling to semi-automatically extract microservices from monolithic codebases based on identified boundaries, handling data migration, shared state, and API design challenges.

Comparative architecture evaluation. Support simultaneous comparison of multiple candidate architectures within a single benchmark, enabling more efficient exploration of the design space.

Advanced decomposition algorithms. Integrate additional clustering approaches (spectral clustering, hierarchical methods, hybrid algorithms) and provide interactive refinement capabilities where architects can manually adjust clusters.

User management and multi-tenancy. Implement authentication, authorization, and multi-tenant isolation to enable collaborative use and deployment as a shared organizational service.

Formal empirical evaluation. Conduct controlled studies to evaluate ServiceSlicer's effectiveness compared to manual decomposition or alternative tools, measuring decomposition quality, user productivity, and scalability prediction accuracy.

CI/CD integration. Embed analysis and benchmarking capabilities within continuous integration workflows to monitor architectural quality as systems evolve, alerting when coupling increases or scalability regresses.

7.5 Closing Remarks

The transition from monolithic to microservices architectures remains one of the most challenging decisions in modern software engineering. Poor decomposition decisions can result in systems that are more complex, less performant, and harder to operate than the monoliths they replaced. ServiceSlicer contributes to addressing this challenge by providing integrated tooling that spans boundary identification, semantic refinement, and empirical validation.

The demonstrations validate the feasibility of the approach and illustrate its application to real systems. While significant work remains—particularly in extending language support, automating code extraction, and conducting formal evaluation—ServiceSlicer represents a step toward making systematic, data-driven monolith decomposition accessible to practitioners. As organizations continue to grapple with architectural modernization, tools that reduce uncertainty, accelerate learning, and provide empirical validation will become increasingly valuable.

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. *O'Reilly Radar*. 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. 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.

14. 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.

15. 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.

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

17. 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.

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