Distributed-to-Centralized: Closing the Loop on Architecture Migration via Unification Refactoring

Provakar Mondal[*†]
Software Innovations Lab, Virginia Tech, USA

Joshua Martin[*]
Software Innovations Lab, Virginia Tech, USA

Eli Tilevich[†]
Software Innovations Lab, Virginia Tech, USA

Abstract

To address business needs and resource constraints, software systems are distributed across multiple execution sites. As requirements evolve and constraints change, distribution may need to be either increased or decreased. While increasing distribution—converting centralized software into distributed components—has been a target of a concerted research effort, decreasing distribution—unifying distributed components—has been largely neglected. Yet recent industry announcements report on reducing distribution by consolidating microservices into monoliths, albeit without a systematic treatment of the associated challenges and solutions. Hence, such migrations rely on manual, ad hoc rewrites that are difficult to plan, execute, and validate. To close the loop on architecture migration, this paper presents Unification Refactoring, which reduces distribution by unifying distributed components into a functionally equivalent centralized system. We realize Unification Refactoring as µjUniter, a proof-of-concept framework that systematically transforms microservice-based architectures into modular monoliths. We evaluated µjUniter on third-party and synthetic microservice systems. Our evaluation demonstrates that µjUniter consistently preserves system functionality, as validated by 409 tests. It also decreases invocation latency, memory consumption, and CPU utilization by ≈65%, ≈72%, and ≈33%, respectively. Furthermore, by automating tedious and error-prone source code transformations, µjUniter minimizes manual intervention, thereby benefiting developers seeking to reduce distribution. By closing the loop on architecture migration, Unification Refactoring addresses a critical omission in the automatic adaptation of modern software.

2012 ACM Subject Classification Software and its engineering → Software system structures; Computer systems organization → Distributed architectures

Keywords and phrases Automated Refactoring, Source Code Transformation, Program Analysis, Microservices, Monoliths

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2026.24

1 Introduction

Software systems continuously adapt their deployment boundaries to align with evolving operational demands, fluctuating resource constraints, and the ongoing necessity of maintenance [36]. Routine maintenance often ranges between small-scale changes to fundamental architectural transformations [47]. Some of these transformations include adapting the system's distribution levels to cope with evolving resources and requirements. As software systems grow in size and complexity, distribution adaptation has become increasingly difficult and costly [86].

[*] Equal Contribution
[†] Corresponding Author

Most modern software systems follow one of the following two architectural styles: (1) monoliths—larger centralized systems, or (2) microservices—distributed systems comprising smaller independent services [3]. Microservices have gained substantial popularity due to the growth of cloud and edge computing, thus increasing the need for larger integrated systems [71]. Despite the demonstrated benefits of microservices in many modern deployments, this architecture may not remain the most suitable as the system faces new requirements [43]. New operational realities often bring to the forefront known downsides of microservices, including increased management complexity, higher latency due to network communication, and additional deployment and monitoring overhead [75]. One way to address these downsides is to transition the system from microservices to a monolithic architecture [25].

Hence, modern system architectures are being migrated from monoliths to microservices and vice versa. The research community, however, has primarily focused on transforming monoliths into microservices, having advanced numerous state-of-the-art program analyses and transformations [32, 60, 39, 48]. In contrast, it has largely neglected the reverse journey.

Nevertheless, several recent industry announcements describe experiences of migrating their software architectures from microservices to monoliths [41, 8, 10]. Notwithstanding their fascinating experiences, these reports neither provide a systematic account of the challenges involved nor describe a step-by-step process for performing the transformation. Nor do they mention any automated tools that facilitated the migration. We surmise that these state-of-the-practice efforts relied largely on manual rewrites that are difficult to plan, execute, and verify. A systematic formulation of this migration as a refactoring—a sequence of structural changes that preserve functionality—would yield new insights into this architectural transformation and its challenges. Once codified, it also enables an investigation into how the process could be automated.

This work aims to close this loop, enabling automated refactoring not only to increase distribution but also to decrease it as necessary. Specifically, we present a new refactoring technique—Unification Refactoring, that transforms distributed components into a semantically equivalent centralized component. We realize our approach as µjUniter, a proof-of-concept refactoring framework that takes as input **micro(μ)**-services, written using the **J**ava Spring Boot framework, and **unites** their source code into a semantically equivalent monolith. To accomplish its automatic refactoring, µjUniter introduces novel program analyses and code transformation routines that automate the core structural migration, thereby minimizing the manual effort required to migrate microservices to monoliths. We evaluate µjUniter by applying it to third-party and synthetic microservice systems, demonstrating the preservation of system functionality, reduced resource consumption, and lowered service invocation latency. We also report on the automation value of µjUniter, demonstrating that it eliminates the need for manual rewrites of communication and some failure-handling logic. Consequently, developer effort is shifted from low-level code refactoring to a modest oversight role in policy consolidation.

The contributions of this paper are as follows:

1. Unification Refactoring, an architectural refactoring for transforming the source code of distributed components into a functionally equivalent centralized component.
2. µjUniter, an automated framework that reifies Unification Refactoring for the domain of Java Spring Boot microservices.
3. An empirical evaluation of µjUniter that demonstrates the effectiveness of the automated refactoring transformations and their impact on system performance and resource consumption.

The rest of this paper is structured as follows. Section 2 provides the technical back-

ground and motivation for this work; Section 3 describes the system design of Unification Refactoring, its main components, and workflow; Section 4 describes how we implemented µjUniter—a proof-of-concept of our approach to automatically uniting disparate microservices into a semantically equivalent monolith; Section 5 presents how we evaluated various characteristics of µjUniter; Section 6 discusses the applicability and limitations of our approach; Section 7 describes the related state of the art; finally, Section 8 concludes the paper with future work directions.

2 Background and Motivation

In this section, we describe the technical concepts used in this work and present real-world cases that motivate this research.

2.1 Background

The technical background of this research includes the concepts of refactoring and the architectural styles of microservices and monoliths, which we outline in turn.

Refactoring

In its classic definition, *refactoring* is a semantics-preserving program transformation intended to improve source code quality, making it easier to maintain and extend [24]. Refactoring has been applied to improve various aspects of source code and program execution [1], such as increasing energy efficiency [56], bolstering security [2], adopting new language features [78, 40], and migrating to new execution environments [44]. Meanwhile, most contemporary software systems are distributed, with different parts of their execution running on remote machines and communicating over a network. Hence, some refactoring techniques, such as *partitioning* and *offloading*, transform a centralized system into a distributed execution model [80, 45, 49]. However, while the literature extensively covers the decomposition of software into distributed components, the inverse transformation—systematically unifying distributed source code to reduce distribution—remains an unexplored frontier in automated refactoring.

Microservices and Monoliths

The microservices architecture organizes a system into loosely coupled services that communicate over a network, commonly via RESTful APIs, enabling independent scaling and fault isolation [81]. This architecture manages microservices via a combination of components and strategies, including API gateways, load balancers, and other mechanisms. This architecture promotes the independent development, deployment, and scaling of individual services, making it well-suited for large-scale distributed systems [61]. However, the distributed nature of microservices can incur significant operational complexity, including challenges in service-to-service communication, deployment orchestration, fault tolerance, and data consistency [75].

In contrast, a monolithic architecture consolidates functional units into corresponding modules deployed as a centralized system, thereby enabling modules to interact via local function calls [64]. While monoliths may offer less scaling flexibility, they can simplify deployment pipelines, reduce inter-component communication latency, and lower operational overhead [76].

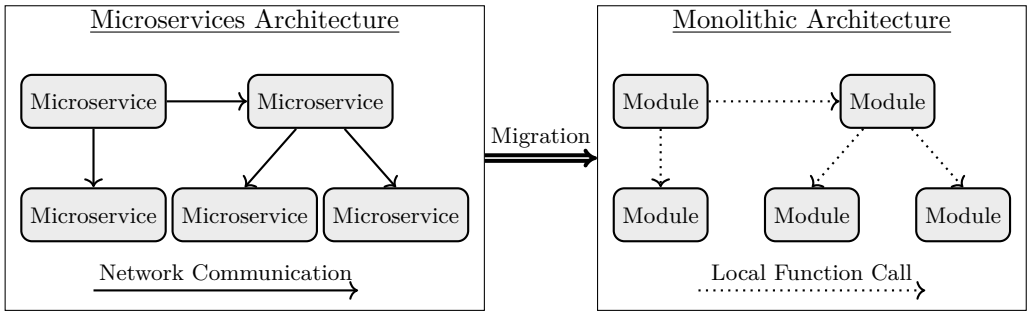


Figure 1 Migration from Microservices to Functionally Equivalent Monolithic Architecture

Figure 1 depicts the architectural migration from a microservices-based system, with separate microservices communicating with each other over a network, to an equivalent monolithic system, in which the original microservices become corresponding local modules, communicating with each other via local function calls.

2.2 Motivating Industry Announcements

We respond to the observation that enterprise systems sometimes need to adjust their distribution levels. Consider the *comment* to a YouTube clip on Amazon Prime Video's migration from microservices to a monolith: *"been in IT for 30 years. it is an endless cycle of flipping back and forth on architecture decisions. can't remember anymore if we are centralizing or decentralizing now"* [28]. As evidenced by this comment's popularity, it expresses a shared sentiment among software developers. Our work is inspired by several prominent enterprises that, for various reasons, needed to reduce the distribution of their systems, in some cases migrating from microservices to monoliths, and subsequently released announcements reflecting on their motivations and experiences.

2.2.1 Prime Video

One announcement reports that Prime Video's migration of a stream monitoring application reduced costs by as much as 90% [41]. As their initial serverless system grew, the team observed an unexpected spike in costs and new scaling bottlenecks. Analysis revealed that the culprits were the orchestration workflow and inter-service communication. While the announcement refers to consolidating microservices, the resulting architecture was not fully monolithic [35]. However, the migration reduced distribution and sparked many subsequent discussions about the real-world effects of overly distributed system overheads [9, 28].

2.2.2 Twilio Segment

Another announcement concerns the customer data platform—Segment. The primary function of Segment's enterprise application is to interact with numerous external data providers. To improve fault isolation, the application was initially migrated to a microservices architecture. However, the resulting system did not meet expectations, largely due to the disadvantages of microservice complexity, prompting a reversal of the journey [8]. Specifically, the distributed microservice architecture assigned one service per provider; thus, as the number of providers increased, the number of services also increased. This growth eventually led to an inordinate increase in operational overheads and development time. Having tried various solutions to

address these issues, the development team determined that excessive distribution was the primary cause and that the best option was to revert to the monolith.

2.2.3 Istio

The last announcement concerns Istio—a popular open-source service mesh used to manage and run microservice applications [51]. This case is especially consequential: given the nature of their service, the development team possessed profound expertise in microservices, yet they decided to migrate their service to a monolith. The announcement first notes that microservices are beneficial only when the value of independent delivery teams, rollouts, and scaling exceeds the cost of orchestration. Since Istio's control plane did not reap the benefits of microservices, it was decided that the original components needed to be consolidated into a monolith. This migration facilitated installation, configuration, use of VMs, maintenance, scalability, and debugging. Furthermore, the development team reported that the monolith reduced startup time and resource usage, resulting in improved responsiveness.

These announcements report on migrations performed on proprietary systems; therefore, source-code-level details cannot be independently verified. Nevertheless, the described business and operational realities firmly establish the necessity to reduce distribution in some cases. Furthermore, none of the narratives mentions any automated tools used to facilitate the migrations. Since distribution must be increased and decreased, the migration toolchain must be extended accordingly. Hence, we view their experiences as motivation to develop an automated refactoring that can serve as a blueprint for similar undertakings.

Our Perspective

We do not advocate for the superiority of either microservices or monolithic architectures, as each possesses well-known advantages and limitations. In this paper, we describe how the microservices-to-monolith migration can be framed as an automated source code refactoring, and we evaluate the effectiveness of the migration and its impact on performance and resource consumption. Finally, this work focuses solely on automating the source-code refactoring required for the migration, assuming that transforming a microservices system into a monolith has already been established as beneficial.

3 System Design and Workflow

In this section, we describe the system design and workflow of Unification Refactoring. As an architectural refactoring, it describes a large-scale transformation that modifies the software architecture while preserving the business functionality. Distributed system architectures come in a wide variety, but they can benefit from Unification Refactoring when the degree of distribution needs to be reduced. Since refactoring is expected to preserve the system's software quality, Unification Refactoring needs to maintain the modularity and structural advantages of distributed components while eliminating the inherent communication and orchestration overheads. Hence, it primarily focuses on replacing remote inter-component communication with local function calls. We demonstrate the main components of Unification Refactoring by transforming a microservice system (Before) into a modular monolith (After), where each original service is preserved as a distinct module, connected via local calls, with a unified entry point. Figure 2 depicts these components and the general flow between them, which we describe next.

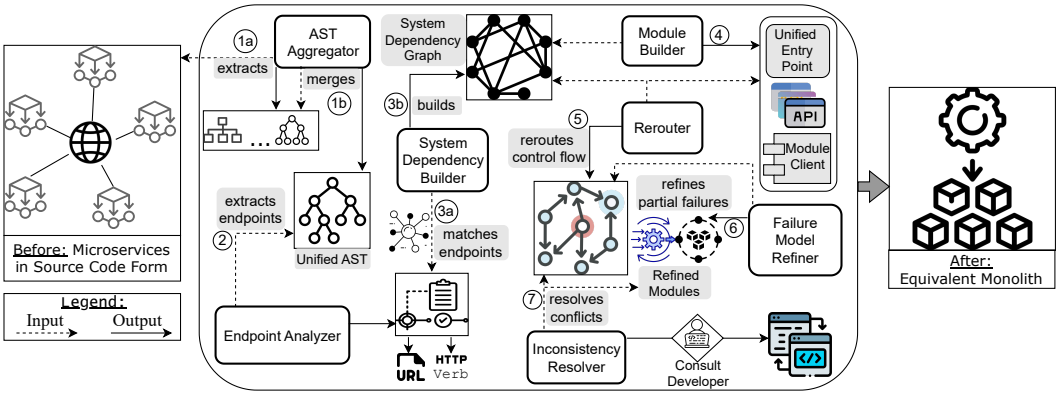


Figure 2 UNIFICATION REFACTORING: Main Components and General Flow

3.1 Abstract Syntax Tree (AST) Aggregator

An Abstract Syntax Tree (AST) is a common representation of a program's source code used for various language processing tasks, such as static analysis and program transformation [59]. In this structured tree-based representation, each node corresponds to a syntactic construct, such as a class, method, or statement. AST preserves the code's syntactic and structural information, while abstracting low-level details. Having received the microservice source code, the *Aggregator* first extracts the AST for each microservice, (1a). The *AST Aggregator* then merges the ASTs of all microservices into a single unified AST, (1b). The aggregation strategy maintains service boundaries by placing each service into a separate module. In cases where duplicate names arise among the aggregated microservices, naming clashes are resolved by renaming the duplicates. Transforming at the AST level enables reasoning about and manipulating code in a language-aware way, ensuring that transformations preserve semantics and produce syntactically correct output.

Formally, let the set of microservices be $\mu\mathcal{S} = \{\mu S_1, \mu S_2, \ldots, \mu S_n\}$ and their ASTs be $\mathcal{A} = \{AST_1, AST_2, \ldots, AST_n\}$ with $AST_i \leftarrow \text{Extract}(\mu S_i)$. The aggregation is defined as: $AST_{\mathcal{U}} = \bigoplus_{i=1}^{n} AST_i$.

3.2 Endpoint Analyzer

Once the unified AST ($AST_{\mathcal{U}}$) is in place, the *Endpoint Analyzer* traverses each module to identify its server endpoints (interfaces the module provides) and client endpoints (interfaces it interacts with). Endpoints include attributes such as URL, HTTP method (POST, PUT, etc.), and payload (JSON, XML, or plain text). The *Endpoint Analyzer* records the location of each endpoint in the AST for later transformation, (2).

For each module M_i in $AST_{\mathcal{U}}$, identify:

$$E_i^{\text{server}} = \{(\text{url}(e_s), \text{verb}(e_s), \text{payload}(e_s)) \mid e_s \in AST(M_i),\ e_s \text{ is a server endpoint}\}$$

$$E_i^{\text{client}} = \{(\text{url}(e_c), \text{verb}(e_c), \text{payload}(e_c)) \mid e_c \in AST(M_i),\ e_c \text{ is a client endpoint}\}$$

3.3 System Dependency Builder

This component builds a System Dependency Graph (SDG) from the endpoint information. An SDG is a graph in which nodes represent modules (former microservices), and edges represent calls from client endpoints to their corresponding server endpoints. By matching

endpoints based on their attributes (URL, HTTP verb, and payload), (3a), this *Builder* constructs the SDG that captures the complete set of inter-service communication pathways, (3b). This graph serves as a blueprint for the subsequent transformation, ensuring that all relevant communication will be rewired into local calls. SDG also provides a global architectural view that can be visualized for examination and debugging before refactoring.

Define the internal communication set:

$$\text{IntComm} = \{(e_c, e_s) \mid e_c \in E_i^{\text{client}}, e_s \in E_j^{\text{server}}, i \neq j,\ \text{Match}(e_c, e_s)\}, \text{where}$$

$$\text{Match}(e_c, e_s) \Rightarrow \text{url}(e_c) = \text{url}(e_s)\ \wedge\ \text{verb}(e_c) = \text{verb}(e_s)\ \wedge\ \text{payload}(e_c) = \text{payload}(e_s).$$

The condition $i \neq j$ excludes intra-module calls, which are already local. Unification Refactoring specifically targets transforming distributed (remote) calls into centralized (local) ones. The System Dependency Graph is then defined as: $SDG = (V, E)$, where

$$V = \{M_1, \ldots, M_n\}\ \wedge\ E = \{(M_i, M_j) \mid \exists (e_c, e_s) \in \text{IntComm}, e_c \in M_i, e_s \in M_j\}.$$

3.4 Module Builder

The SDG in the previous step contains all the modules of the future monolith. However, these modules must be implemented in code, which is the purpose of the *Module Builder* component. It begins by establishing a centralized entry point for the monolith. In a distributed microservice system, each service maintains its own entry point and initialization logic, along with configuration for functionality such as security and integration with external systems. To unify these microservices, this *Builder* consolidates their entry points into a single main entry point that represents the entire system. This unified entry point preserves all essential configurations and behaviors from each constituent service. It also ensures that they coexist without conflict within a unified monolith. As a result, the resulting system can act as a single cohesive unit that preserves the semantics of the original microservices.

With the main entry point complete, the *Builder* creates a Module API for each module, thereby exposing its functionality to other modules. This step also bridges the control flow between modules, beginning with the server endpoint's location within the AST. These APIs expose a proxy that handles incoming requests and prepares responses, enabling modules to communicate with one another by calling these APIs.

Once the APIs are in place, the *Builder* produces Module Clients for the modules that invoke these APIs. Each client encapsulates the logic required to invoke another module's API, taking over the role that remote communication played in the microservice architecture. In addition to calling the API methods, module clients handle data adaptation to maintain compatibility when different modules use distinct but structurally similar data representations. In a distributed system, such adaptation often occurs implicitly during serialization and deserialization [79]; in a monolith, it must be handled explicitly to ensure that data exchanged between modules remains correct, consistent, and isolated from unintended side effects [53]. This explicit adaptation is also required to prevent unintended coupling between modules' internal states, so that they remain logically separated, as in the original distributed system.

In summary, the *Module Builder* constructs the unified entry point, module API, and module clients (4) for the resulting monolith. Formally, all microservice-specific entry points $\{EP_1, EP_2, \ldots, EP_n\}$ are consolidated into a single unified entry point $EP_{\mathcal{U}}$, preserving their initialization logic and configurations:

$$EP_{\mathcal{U}} = \text{Unify}(EP_1, EP_2, \ldots, EP_n).$$

Then, for each server endpoint $e_s \in E_i^{\text{server}}$, the *Builder* creates an API method $API_i(e_s)$, which exposes the corresponding functionality as an internal callable:

$$API_i = \{f_{e_s} \mid e_s \in E_i^{\text{server}}\}, \quad f_{e_s} : \text{payload}(e_s) \mapsto \text{response}(e_s).$$

Finally, for each client endpoint $e_c \in E_i^{\text{client}}$ that matches a server endpoint $e_s \in E_j^{\text{server}}$ (with $i \neq j$), the *Builder* generates a client method $Client_i(e_c)$ that invokes the corresponding API method f_{e_s} while applying an adaptation function α if the payload or response structures differ:

$$Client_i(e_c) = \lambda x.\ f_{e_s}(\alpha(x)), \text{where} \ \ \alpha : \text{payload}(e_c) \to \text{payload}(e_s).$$

3.5 Rerouter

The *Rerouter* component replaces previously invoked remote endpoints of microservices with calls to the corresponding local module methods, ⑤. The original microservices used remote communication across the network. To reroute control flow from this remote communication over the network to local function calls, the *Rerouter* replaces all call sites that initiate the remote communication with the newly built module clients. The rerouting transforms distributed control flow into local, intra-process control flow.

Formally, after rerouting, the client of module M_i invokes the API of module M_j that corresponds to the server endpoint e_s of the original microservice system:

$$\forall (e_c, e_s) \in \text{IntComm}: \quad \underbrace{\text{Call}(e_c, x)}_{\text{remote}} \xRightarrow{\text{Reroute}} \underbrace{\text{Call}\big(f_{e_s}, \alpha(x)\big)}_{\text{local, intra-process}}, \quad f_{e_s} = API_j(e_s),$$

where $x \in \text{payload}(e_c)$ denotes the argument (payload data) passed at the original client call site, and α is the adaptation function that maps the payload expected by the client endpoint e_c into the payload schema required by the server endpoint e_s.

3.6 Failure Model Refiner

To preserve semantic equivalence, Unification Refactoring must ensure that the architectural migration accounts for the disparate failure models of distributed and centralized executions. Distributed systems are prone to *partial failures*, conditions under which a subcomponent becomes unreachable or inoperational while the rest of the system continues to function normally [55]. In contrast, the failure model of monoliths is *total* rather than *partial*. In other words, a monolithic system can also experience faults in its business logic and other local components, and if these are not recoverable, the system terminates [52]. The failures that pertain to distributed execution can no longer occur once the distributed components have been consolidated into a monolith, so the logic for handling such failures becomes superfluous. To remove this now-unnecessary logic, Unification Refactoring employs a *Failure Model Refiner*, whose high-level functionality is described next, ⑥.

Formally, let e_s be a server endpoint and $x \in \text{payload}(e_s)$. In a distributed environment, the result of an invocation by e_c, denoted $\mathcal{O}_{dist}$:

$$\mathcal{O}_{dist}(e_c, x) \in \{\text{response}(e_s)\} \cup \mathcal{E}_{app} \cup \mathcal{E}_{dist}$$

where $\mathcal{E}_{app}$ represents application-level exceptions (e.g., business logic errors) and $\mathcal{E}_{dist}$ represents distribution-induced failures (e.g., timeouts, network partitions).

By unifying distributed components, Unification Refactoring eliminates the network as a source of failure. The refactoring process maps the original exception handlers H

via a transformation function $\mathcal{T}(H)$. If H catches an element of $\mathcal{E}_{dist}$, it is flagged as *distribution-induced dead code*, which should either be pruned entirely or perhaps substituted with local fallback logic. Consequently, the monolithic outcome $\mathcal{O}_{mono}$ is a refinement where $\mathcal{O}_{mono} = \mathcal{O}_{dist} \setminus \mathcal{E}_{dist}$. Hence, the role of the *Failure Model Refiner* is to remove only the failure-handling logic that is no longer possible due to the replacement of distributed communication with local functions. The original business logic, which should remain intact, can experience other failures, whose handling logic must be preserved.

3.7 Inconsistency Resolver

Finally, the *Inconsistency Resolver* component addresses cross-cutting consolidation issues that arise from unifying multiple microservices into a monolith, ⑦. The resolution involves resolving naming conflicts, internalizing API gateway routes to prevent endpoint collisions, and merging security policies so that the monolith enforces the strictest applicable rules. This step ensures that the monolith is both functionally correct and operationally coherent. Unfortunately, the resolution does not easily lend itself to complete automation, as many of these non-functional concerns are quite domain-specific. For example, security policies are known to be defined and enforced in ways that vary widely across applications and even deployments. Hence, both manual examination and correction are required.

At the end of this process, the refactoring produces a modular monolith whose external behavior is indistinguishable from that of the original microservice system, but which executes without inter-service network calls. The transformed system continues to adhere to well-established software engineering principles, including high cohesion, low coupling, and statelessness. The refactoring transformations typically reduce aggregate invocation latency by minimizing communication overhead, thereby making remote inter-service communication equivalent to local inter-module calls. Additionally, orchestration overhead decreases due to the reduced need for service discovery, gateways, internal traffic management, and other management services.

Formally, the resulting monolith is defined as:

$$\mathcal{M} = \big(\{M_1, \ldots, M_n\},\ EP_{\mathcal{U}}\big),$$

where each module $M_i = (API_i, \{Client_i(e_c) \mid e_c \in E_i^{\text{client}}\})$ corresponds to a former microservice μS_i, and all inter-service communication is transformed from distributed network calls into intra-process function calls. Interaction between modules is then expressed as:

$$\forall e_c \in E_i^{\text{client}}, e_c \mapsto e_s \in E_j^{\text{server}}, x \in \text{payload}(e_c) :$$
$$M_i.\,Client_i(e_c)(x) \ \rightarrow\ M_j.\,API_j(e_s)(x) \in \mathcal{O}_{mono}(e_c, x)$$

where $\mathcal{O}_{mono}(e_c, x) = \{\text{response}(e_s)\} \cup \mathcal{E}_{app}$. By explicitly excluding $\mathcal{E}_{dist}$ (fault-handling logic required for distribution-induced failures), Unification Refactoring fully transforms a distributed inter-component communication into semantically equivalent intra-process execution.

4 Implementation: µjUniter

As a concrete implementation of Unification Refactoring, we created µjUniter that targets microservice applications built with the Java Spring Boot framework [7]. Both empirical and practical reasons motivate this choice. On the empirical side, microservices are mainstream in industry: 37% of programmers report working with microservices, and 34% of

them use Java to do so, according to the 2022 JetBrains Developer Ecosystem survey [37]. Within the Java ecosystem, Spring Boot dominates microservice development, with 59% of Java developers reporting that they use this framework, according to the 2023 JRebel Java Productivity Report [67]. These statistics highlight that Java and Spring Boot are not only widely used but also representative platforms for enterprise-grade microservices. On the practical side, Spring Boot's heavy reliance on annotations and declarative configurations provides a non-trivial and illustrative testbed [57]. Hence, we decided to focus on Spring Boot microservices as the target of our implementation. By demonstrating the ability to unify these microservices, µjUniter's design reveals implementation insights that can be adapted to other languages and frameworks. Recall that the system design of Unification Refactoring comprises seven main components, whose implementation in µjUniter is described next.

4.1 AST Aggregator

For all source code analysis and transformation tasks, we used the Spoon Java library [66]. This library parses each microservice's source code into a Spoon's `CtModel` AST, which provides an API for traversing, editing, and regenerating source code. For each extracted microservice AST, the *Aggregator* resolves naming conflicts at the package and class level and aggregates them into a single unified tree.

This aggregation requires certain nuance, as a naïve aggregation of all microservice root packages into one may create conflicting fully qualified names. Such conflicts may cause undefined behavior later, when the AST is converted back into source code. Hence, the *Aggregator* must ensure that the fully qualified names of all classes are unique. Possible solutions include class renaming [72], modular packaging[42], or various modularization approaches such as Java 9 modules [14], Maven Modules [23], or Spring Modulith [17]. While the modularization approaches provide the best encapsulation, they introduce new dependencies, increase complexity, and are inconsistent across programming languages [5]. For this reason, we use modular packaging, placing each service's source code in a package named after the service. This step first places each service into a uniquely named package, then aggregates all packages into a single AST.

This co-location of all code in a single representation is essential for whole-system analysis, enabling the detection of cross-service relationships and preventing high coupling in the services' internal logic. Algorithm 1 shows the step-by-step pseudo-code of the AST aggregation task.

4.2 Endpoint Analyzer

For this component, we implement AST traversal routines that locate service endpoints by detecting Spring-specific Java annotations. Server endpoints are identified through annotations such as `@RestController` and `@RequestMapping` (including derived forms like `@GetMapping` and `@PostMapping`), while client endpoints are identified via annotations such as `@FeignClient`. These annotations and their programming conventions serve as reliable markers for identifying endpoints. For each endpoint, we record not only the metadata (URL, HTTP verb, payload) but also the exact AST node location, ensuring that transformations in later stages can directly modify or replace the corresponding methods. This annotation-driven analysis shows how tightly implementation details are coupled to framework conventions, and reinforces that support for other ecosystems would require re-implementing this analysis step with their respective idioms.

Algorithm 1 AST Extraction and Aggregation

Input: $\mu S = \{\mu S_1, \ldots, \mu S_n\}$: Set of Input Microservices
Output: $AST_{\mathcal{U}}$: Unified Abstract Syntax Tree

$\mathcal{A} \leftarrow \emptyset$
forall $\mu S_i \in \mu S$ **do**
  $AST_i \leftarrow$ Spoon.ExtractAST(μS_i)
  $AST_i \leftarrow$ ApplyModularPackaging(AST_i, packageName = μS_i.name)
  /* Compare with previously aggregated ASTs to ensure unique class names */
  $AST_i \leftarrow$ ResolveNamingConflicts($AST_i, \mathcal{A}$)
  $\mathcal{A} \leftarrow \mathcal{A} \cup \{AST_i\}$
$AST_{\mathcal{U}} \leftarrow$ AggregateASTs($\mathcal{A}$)
return $AST_{\mathcal{U}}$

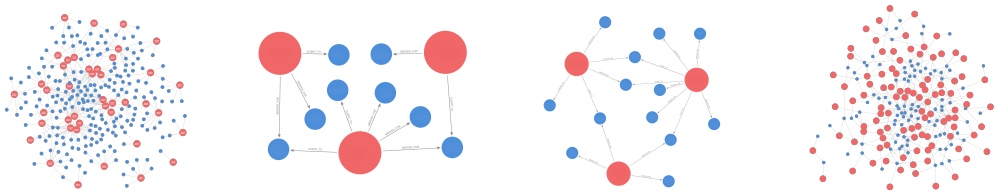
4.3 System Dependency Builder

This component constructs a System Dependency Graph (SDG) using Algorithm 2 by matching endpoints in the unified AST. The built SDGs are stored in Neo4j [46], a graph database, via the `NeoRepository class`. Neo4j enables not only persistence but also interactive exploration through tools such as Neo4j Bloom [34]. Before proceeding with a refactoring transformation, a developer may want to validate that all inter-service communication paths are captured. Although visualizations are not strictly necessary to implement µjUniter, modern developers expect refactoring tools to facilitate understanding and validation. As an example, Figure 3 visualizes the built SDGs for our evaluation subjects prior to their unification. We detail the subjects in Section 5.1. In these diagrams, the red and blue circles represent the microservices and their endpoints, respectively.

Algorithm 2 Endpoint Extraction, Matching, and SDG Construction

Input: $AST_{\mathcal{U}}$: Unified AST
Output: $SDG = (V, E)$: System Dependency Graph
Output: IntComm: Internal Communication Set

$V \leftarrow \{M_1, \ldots, M_n\} \leftarrow$ ExtractModules($AST_{\mathcal{U}}$)
IntComm $\leftarrow \emptyset$
foreach $M_i \in V$ **do**
  $E_i^{server} \leftarrow$ LocateEndpoints(M_i, {`@RestController`, `@RequestMapping`, ...})
  $E_i^{client} \leftarrow$ LocateEndpoints(M_i, {`@FeignClient`})
foreach $e_c \in E_i^{client},\ e_s \in E_j^{server},\ i \neq j$ **do**
  if url(e_c) = url(e_s) $\wedge$ verb(e_c) = verb(e_s) $\wedge$ payload(e_c) = payload(e_s) **then**
    IntComm $\leftarrow$ IntComm $\cup \{(e_c, e_s)\}$
$E \leftarrow \{(M_i, M_j) \mid \exists (e_c, e_s) \in \text{IntComm},\ e_c \in M_i, e_s \in M_j\}$
$SDG \leftarrow (V, E)$
return (SDG, IntComm)



(a) Train Ticket **(b)** Piggy Metrics **(c)** SCShowcase **(d)** RM-100-0.02

Figure 3 System Dependency Graph Exemplars

4.4 Module Builder

This component performs three actions: (1) creates a unified main entry point, (2) creates Module APIs, and (3) builds Module Clients. (1) A new main class is created to consolidate the entry points of all services. This unified class merges `@SpringBootApplication` annotations, initialization logic, and configuration beans, while resolving conflicts through systematic renaming and refactoring. In Java, this requires preserving all in-code configurations from each service's main class, including annotations, beans, and methods for enabling clients, defining APIs, implementing security, and configuring repositories. The result is a single monolithic entry point that preserves the startup semantics of all original microservices. (2) Module APIs are generated for accessing module functionality. These APIs act as proxies for controller methods, creating explicit entry points for inter-module calls and defining a controlled location for inter-module communication. (3) Module Clients are generated for invoking module APIs. Algorithm 3 details these three actions.

Algorithm 3 Module Building for Unified Entry Point, APIs, and Clients

Input: $AST_{\mathcal{U}}, SDG, \text{IntComm}$
Output: $\mathcal{M} = \{M_1, \ldots, M_n\}$: Set of Monolith Modules

/* unifying microservice-specific entry points (EPs) into one */
$EP_{\mathcal{U}} \leftarrow \text{Unify}(EP_1, \ldots, EP_n)$
$\mathcal{M} \leftarrow \emptyset$
foreach $M_i \in SDG.V$ **do**
  $API_i \leftarrow \{f_{e_s} \mid e_s \in E_i^{server}\}$, with $f_{e_s} : \text{payload}(e_s) \mapsto \text{response}(e_s)$
  $Client_i \leftarrow \emptyset$
  foreach $(e_c, e_s) \in \text{IntComm},\ e_c \in M_i$ **do**
    /* $\alpha : \text{payload}(e_c) \to \text{payload}(e_s)$ is the adaptation function */
    $Client_i(e_c) \leftarrow \lambda x.\ f_{e_s}(\alpha(x))$
  $\mathcal{M} \leftarrow \mathcal{M} \cup \{M_i(API_i, Client_i)\}$
return $\mathcal{M}$

Each client method invokes the corresponding API method in the target module and also provides an additional routine, deep-copy, for type conversion [85]. This routine emulates structural typing within a nominally typed language for the microservice parameters and return types that are Java type-incompatible. Accommodating such type-incompatibility requires generating adaptation logic. When transferring data across distributed sites, structurally compatible but type-unrelated objects (parameters and return types) can be serialized into a single type and then unserialized into another [68]. When the distribution is removed, this component emulates serialization via deep copying. Specifically, deep-copy routines are

generated for all pairs of incompatible types whose objects need to be assigned to each other.

The deep copy is generated as follows. Immutable objects are passed directly, iterables are reconstructed item by item, and custom models are recreated via constructors and accessors, with fields copied using getter and setter methods. Because server endpoints can serve multiple clients with different expectation types, the copy-conversion logic is generated in the client rather than the API class. If automated copying fails, serialization is retained as a conservative fallback.

4.5 Rerouter

This component replaces the remote calls in client modules with local calls to the generated Module Client methods. The replacement is performed via AST rewriting: each method invocation node is replaced with a new node corresponding to the local client call, preserving its arguments and signature. Spoon's ability to query the AST for invocations by annotation and type makes this possible without fragile text-based pattern matching, such as scanning source files for occurrences of `RestTemplate.getForObject(...)` or `FeignClient` method calls, or searching for string literals like `"http://<service>/<endpoint>"`. This rerouting step completes the replacement of network calls with local calls. Algorithm 4 shows the step-by-step pseudo-code of the control flow rerouting task.

Algorithm 4 Control Flow Rerouting

Input: $\mathcal{M}$, IntComm
Output: $\mathcal{M}_{rerouted}$: Modules with Local Method Invocations

```
/* Initialize the rerouted set with the original unified modules */
```
$\mathcal{M}_{rerouted} \leftarrow \mathcal{M}$
foreach $(e_c, e_s) \in$ IntComm **do**
  $M_i \leftarrow$ module owning $e_c \in \mathcal{M}_{rerouted}$
  $M_j \leftarrow$ module owning $e_s \in \mathcal{M}_{rerouted}$
  `/* Rewrite AST: replace Feign/RestTemplate calls with direct local calls */`
  $M_i' \leftarrow \text{ReplaceInvocationAST}(e_c, \text{call} = M_j.API_j(e_s), \text{args} = \alpha(\text{payload}(e_c)))$
  $\mathcal{M}_{rerouted} \leftarrow (\mathcal{M}_{rerouted} \setminus \{M_i\}) \cup \{M_i'\}$
return $\mathcal{M}_{rerouted}$

4.6 Failure Model Refiner

In the Java Spring Boot ecosystem, partial failures are typically managed through three primary mechanisms: (1) low-level network exceptions from the `java.net` hierarchy, (2) declarative resilience patterns such as Spring Cloud Circuit Breaker or `Resilience4j`, and (3) HTTP status code mapping via `FeignException` or `HttpStatusCodeException`. To implement the transformation function $\mathcal{T}(H)$ introduced in Section 3.6, µjUniter systematically identifies these artifacts within the Spoon AST and determines if they represent business logic or distribution remnants.

µjUniter identifies distribution-induced failures ($\mathcal{E}_{dist}$) by scanning the unified AST for specific types and annotations that lose their semantic usefulness in a local context. These include connection failures (e.g., `ConnectException`, `SocketTimeoutException`), framework-specific RPC exceptions (e.g., `ResourceAccessException`, `RetryableException`), and resilience annotations (e.g., `@CircuitBreaker`, `@Retry`). Once identified, µjUniter applies domain-specific

Algorithm 5 Failure Model Refinement and Handler Transformation

Input: $\mathcal{M}_{rerouted}$
Output: $\mathcal{M}_{refined}$: Modules with transformed failure handling

$\mathcal{E}_{dist} \leftarrow \{$`ConnectException`, `FeignException`, `TimeoutException`, $\dots\}$
$\mathcal{A}_{dist} \leftarrow \{$`@CircuitBreaker`, `@Retry`, `@Bulkhead`$\}$
$\mathcal{M}_{refined} \leftarrow \mathcal{M}_{rerouted}$ `// Initialize with rerouted modules`
foreach $M_i \in \mathcal{M}_{refined}$ **do**
  `/* Identify and refine resilience annotations (`$\mathcal{A}_{dist}$`) */`
  $nodes \leftarrow$ Spoon.findAnnotated$(M_i, \mathcal{A}_{dist})$
  foreach $n \in nodes$ **do**
    if $n.hasFallback()$ **then**
      ApplyBusinessFallback(n) `// Preserve functional defaults`
    Spoon.removeAnnotation(n)
  `/* Transform exception handlers` H `via function` $\mathcal{T}(H)$ `*/`
  $handlers \leftarrow$ Spoon.findTryCatchBlocks(M_i)
  foreach $H \in handlers$ **do**
    if $H.getCaughtType() \in \mathcal{E}_{dist}$ **then**
      $sideEffects \leftarrow$ AnalyzeSideEffects(H)
      if $sideEffects \in \{Logging,\ Retry\}$ **then**
        $\mathcal{T}(H) \leftarrow \emptyset$ `// Prune distribution-induced dead code`
      else
        $\mathcal{T}(H) \leftarrow$ RefactorToLocal(H) `// Map to application logic`
  `/* Update` M_i `and remove obsolete checked exception signatures */`
  $M_i' \leftarrow$ CleanupCheckedExceptions$(M_i,$ `"java.io.IOException"`$)$
  $\mathcal{M}_{refined} \leftarrow (\mathcal{M}_{refined} \setminus \{M_i\}) \cup \{M_i'\}$
return $\mathcal{M}_{refined}$

transformation rules based on the functionality of the original handlers. A significant challenge arises when a developer implements advanced business-logic strategies, such as handling network failures with a local fallback, e.g., returning a default cached object when a service is unreachable. If µjUniter finds a `catch` block for an element of $\mathcal{E}_{dist}$ containing such logic, it preserves the business response but refactors the trigger to a local equivalent. It is worth noting that µjUniter does not always correctly disentangle these distributed and business functionalities, a lack of precision that we consider acceptable for a refactoring tool typically used under the programmer's oversight.

For resilience patterns, µjUniter removes the proxying infrastructure because it imposes unnecessary overhead in the absence of distributed faults. Specifically, maintaining a half-open [77] state for a local method call would introduce unnecessary latency without providing fault isolation. Hence, such patterns need to be removed to ensure efficient local execution. A final practical hurdle is the handling of checked exceptions. Many microservices declare `throws IOException` solely to account for network volatility, a condition that can no longer be present once distributed communication is removed. µjUniter addresses this now-unnecessary exception declaration by performing a *signature cleanup*. It traverses the call graph in the modular monolith and removes distribution-related checked exceptions from method signatures if the new local implementation no longer throws them. This cleanup

ensures that the resulting monolith appears to have been originally engineered for centralized execution. Algorithm 5 details the implementation of this refinement process.

4.7 Inconsistency Resolver

As the final step, this component addresses cross-cutting consolidation issues. For dependency management, we rely on Maven [22], which provides both a build system and a consistent mechanism for reconfiguring the project as a single modular monolith. To address name conflicts in dependency-injection beans, we extend our Spoon-based transformations to rename beans and update their references. For security consolidation, we analyze the Spring Security configurations in each service and merge them into the monolith, defaulting to the strictest applicable policy. Path rewrites introduced by API gateways necessitate endpoint-path normalization to avoid collisions. These tasks highlight that, although architectural refactoring is largely automated, system-specific configurations still require careful handling and, in some cases, human oversight.

This component's functionality can be described as a sequential pipeline of refinements that resolves cross-cutting discrepancies across four distinct tasks:

Task I: Project Reconfiguration ($\mathcal{M}_{refined} \rightarrow \mathcal{M}_{struct}$) By leveraging Maven, this refinement establishes a unified build lifecycle. It consolidates independent build scripts into a multi-module project structure, reconciling top-level dependency management to ensure a consistent classpath across the unified environment.

Task II: Dependency Injection Namespace Resolution ($\mathcal{M}_{struct} \rightarrow \mathcal{M}_{beans}$) To prevent bean name collisions in the shared Spring context, we apply Spoon-based transformations to programmatically rename conflicting beans. All associated injection points and references are updated across the codebase to ensure runtime stability.

Task III: Security Policy Synthesis ($\mathcal{M}_{beans} \rightarrow \mathcal{M}_{sec}$) For each constituent service, the configurations of `WebSecurityConfigurerAdapter` are analyzed to produce a unified security context, defaulting to the strictest applicable CORS/CSRF policies while surfacing complex semantic overlaps for developer oversight.

Task IV: Endpoint Path Normalization ($\mathcal{M}_{sec} \rightarrow \mathcal{M}_{monolith}$) Finally, API gateway routing logic is reconciled by normalizing endpoint paths. Controller mappings are rewritten to include the root paths previously managed by the gateway, preventing URL collisions and ensuring internal communication parity with the original distributed system.

5 Evaluation

The following research questions drive the evaluation of µjUniter:

1. **RQ1: Effectiveness:** How effectively does µjUniter reify Unification Refactoring in transforming microservices into an equivalent monolith?
2. **RQ2: Performance:** How µjUniter's migration affect application performance characteristics?
3. **RQ3: Automation Value:** What is the value of µjUniter automating the required source code transformations?

We begin by describing our evaluation subjects and then present our experimental results, which address the aforementioned questions.

5.1 Evaluation Subjects

Recall that µjUniter targets Spring framework microservice applications. For the evaluation, we prioritized contemporary Spring Boot applications from different domains, including a booking system, a financial advisor, and a cloud platform, that can serve as representative benchmarks. Specifically, we selected the following three third-party Spring applications. Train Ticket [27] and Piggy Metrics [50]—applications whose backends are structured as microservices. The third example, which we refer to as SCShowcase (an abbreviation of SpringCloudShowcase), is a recent microservice tutorial that demonstrates the advantages of Spring Cloud as a microservice platform [54]. To stress-test the applicability of µjUniter to microservice systems containing large numbers of microservices and their interactions, we created a generator that automatically synthesizes such applications. The generator is parameterized by the number of microservices and their interaction probabilities. We describe our evaluation subjects in greater detail next.

Train Ticket

Train Ticket is a dedicated microservice benchmark that implements a complete train-ticket booking system with a web-based front end. This benchmark has been widely used in prior research [82, 11, 74, 12]. The Train Ticket comprises 41 microservices: 37 Java Spring Boot, 2 Python, 1 JavaScript, and 1 Go (incomplete in the repository). Because µjUniter only supports Java Spring microservices, we could not apply it to the other three services implemented in Python and JavaScript. Although we could have excluded them from evaluation, we instead chose to manually translate them to Java because of their relative simplicity and reliance on external libraries with Java equivalents. This system serves as a realistic substitute for an enterprise system, given the complexity of service interactions and the large number and diversity of services.

Piggy Metrics

Piggy Metrics is a financial advisor application. While primarily intended as a tutorial, this system has been used as a benchmark for testing and monitoring microservices [20]. The business logic is contained in the accounts, notifications, and statistics services. The only change made to Piggy Metrics before applying µjUniter was the removal of references to an obsolete external API, causing runtime errors.

SCShowcase

This subject's microservices are implemented in Java 21 using the latest Springboot version. The business logic appears in the employee, department, and organization services. Each service manages its own domain data and exposes REST APIs, while service discovery, an API gateway, and configuration management provide the supporting infrastructure.

RM*

To create complex microservice systems, we developed a Random Microservice Generator, parameterized by the number of microservices and their interaction probabilities. Each generated microservice includes a single controller endpoint and has a certain probability of calling other microservices within the system. Because microservice systems typically

exclude cyclic dependencies [18], the generated microservices must avoid interactions that would create a cycle. To implement this policy, our generator uses a directed acyclic graph.

For our evaluation, we synthesized and unified three microservice systems. RM-10-0.2 is a synthetic microservice system with 10 services and a 0.2 service interaction rate; RM-10-1.0 increases the service interaction rate; and RM-100-0.02 increases the number of services. These properties are specifically designed to stress-test µjUniter with respect to the number of services and endpoints.

Table 1 provides an overview of these evaluation subjects. For some subjects, the number of client and server endpoints differs. This discrepancy arises because some leaf-node services are pure callers with no exposed server endpoints.

Table 1 Evaluation Subjects: Microservice System Details

Subject Name	No. of Microservices	No. of Server Endpoints	No. of Client Endpoints	Endpoint Library
Train Ticket	40	98	157	Spring RESTClient
Piggy Metrics	3	4	4	Spring OpenFeign
SCShowcase	3	13	4	Spring OpenFeign
RM-10-0.2	10	8	14	Spring WebClient
RM-10-1.0	10	9	45	Spring WebClient
RM-100-0.02	100	84	179	Spring WebClient

5.2 RQ1: Effectiveness

For a refactoring to be effective, it must modify the program's structure as intended while preserving the original application's functionality. While we verified that the microservices were transformed into a monolith through a straightforward source-code examination, we sought systematic validation that the functionality was preserved. Although refactoring was originally introduced as "a semantics-preserving transformation," [63], verifying that a semantics is indeed preserved would require first defining it formally. In the absence of formal methods that scale to microservices, we resort to testing as a pragmatic, evidence-based approach to demonstrate functional preservation. In other words, in this evaluation, we verify if the *before* (i.e., original microservices) and *after* (i.e., refactored monolith) pass the same test cases. Of course, Dijkstra's dictum that "testing does not prove the absence of bugs, only their presence" [16] still stands, but testing remains the most widely applied option for verifying the preservation of functionality in practical settings [58, 38].

For our evaluation, we first applied µjUniter to all our evaluation subjects, producing a modular monolith version for each. We then tested both the *before* and *after* versions and verified that the test suites produced the same results. Whenever possible, we used test suites that either came with the evaluation subjects or were adapted from prior research [74]. In the absence of tests, we conducted our own testing of the main functionalities and use cases.

For Train Ticket and Piggy Metrics, we applied end-to-end and unit testing; for SCShowcase, we applied only unit testing because it lacks a front end. Specifically, for these subjects, we executed 340, 49, and 20 test methods, respectively. Our evaluation resulted in (409 / 409) passed tests, meaning that all the testing methods passed in both the *before* and *after* versions. In a way, this experiment can be characterized as *regression testing*, as it runs the same suite against both versions [83]. Table 2 summarizes our evaluation of effectiveness.

Table 2 Testing Scenarios, Statistics, and Outcomes

Use Cases	No. of Methods	Test Types	*Before* Monolith	*After* Microservice
Train Ticket				
End-to-end booking flows (login, search, reserve, pay, rebook)	20	End-to-end (Selenium)	✓	✓
Administrative management features (stations, routes, prices, users)	15	End-to-end (Selenium)	✓	✓
Ticketing and travel operations (orders, payments, cancellations, notifications)	25	End-to-end (Selenium)	✓	✓
Controllers for user-facing REST endpoints (e.g., UserController, TravelController)	120	Unit (controller-level)	✓	✓
Service-layer implementations (e.g., OrderService, PriceService, TravelService)	160	Unit (service-layer)	✓	✓
Piggy Metrics				
Front-end rendering and navigation	1	End-to-end (Selenium)	✓	✓
Account, user, and statistics controllers (API validation)	14	Unit (controller-level)	✓	✓
Core services (AccountService, NotificationService, StatisticsService)	20	Unit (service-layer)	✓	✓
Repositories for persistence (AccountRepository, UserRepository, RecipientRepository)	9	Unit (repository-layer)	✓	✓
External service clients (ExchangeRatesClient and Fallback)	3	Integration (API client)	✓	✓
Email notification functionality (EmailService)	2	Unit (service-layer)	✓	✓
SCShowcase				
Department and Employee domain logic	10	Unit (service-layer)	✓	✓
Organization domain logic and repository	7	Unit (repository-layer)	✓	✓
REST controllers (OrganizationController)	3	Unit (controller-level)	✓	✓

Because we created RM*—our own synthetic microservice applications—with the specific goal of stress testing, they lack coherent business logic. Hence, we could not apply use-case testing to these applications to assess µjUniter's effectiveness. Instead, we performed a two-tier structural verification. First, we conducted `ApplicationContext` loading tests using the `SpringJUnit4ClassRunner` to ensure that all beans were correctly wired across the new modular boundaries. Second, we performed a reachability analysis to ensure that every former remote endpoint was successfully mapped to a local direct method invocation. Table 3 summarizes the results for these synthetic subjects.

Table 3 Structural and Contextual Verification of RM* Monoliths

Subjects	Compilation	Bean Wiring	Context Load	Reachability	Outcome
RM-10-0.2	Successful	112/112 Beans	✓(0.8s)	100%	**Pass**
RM-10-1.0	Successful	148/148 Beans	✓(1.1s)	100%	**Pass**
RM-100-0.02	Successful	1240/1240 Beans	✓(8.4s)	100%	**Pass**

While our functional test suite confirms that µjUniter preserves the core functionality of the migrated applications, these tests are insensitive to the removal of distribution-specific fault handling. Because partial failure mechanisms are orthogonal to business logic, their elimination should not affect observable behavior. Consequently, any residual, unreachable

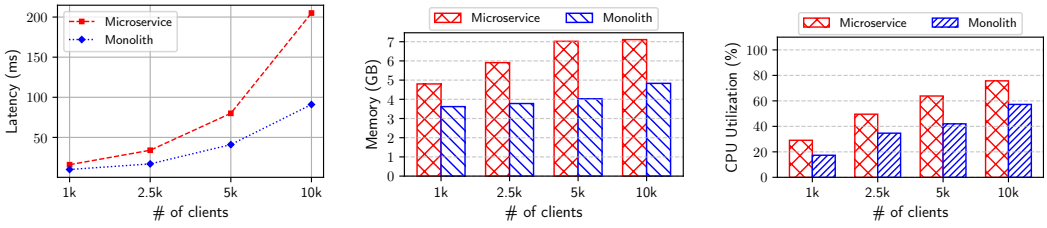
fault-handling code would not compromise the correctness of the application's business logic. Instead, its removal by µjUniter serves as a structural code simplification, reducing technical debt and streamlining execution, rather than a functional requirement. While we have not systematically evaluated µjUniter's dead-code removal, this omission does not impact our primary claim of functional preservation; at worst, any remaining obsolete handlers would exist as harmless, unreachable code that does not affect the application's ability to deliver its functionality.

5.3 RQ2: Performance

We evaluate how Unification Refactoring as realized in µjUniter affects the performance characteristics of our subjects in terms of latency, memory usage, and CPU utilization. We measured the performance for Train Ticket and Piggy Metrics using Gatling [29], a performance-testing framework. Using Gatling, we generated workloads that simulate a specified number of clients accessing the application via the front end. The framework then reports latency, memory consumption, and CPU utilization. For subjects without front ends, we used Apache JMeter [21] to execute workloads. For the synthetic subjects, we kept the number of clients constant (30), as these subjects were used specifically for stress testing with respect to the number of microservices. Using the same workloads, we measured these characteristics for the *before* and *after* versions and compared them.

(a) Latency **(b)** Memory **(c)** CPU Utilization

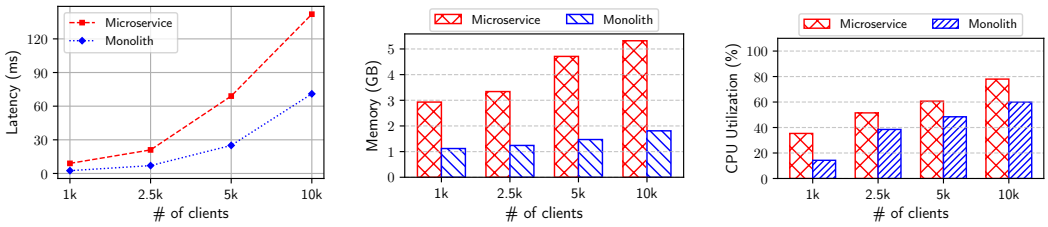
Figure 4 Performance Characteristics for Train Ticket



(a) Latency **(b)** Memory **(c)** CPU Utilization

Figure 5 Performance Characteristics for Piggy Metrics

Figures 4, 5, and 6 depict the performance evaluation results for Train Ticket, Piggy Metrics, and SCShowcase, respectively. Table 4 reports on the measurements for our synthetic microservice subject, RM*. Latency and resource consumption increased proportionally to the number of clients for both versions. However, the monolith versions consistently displayed lower latency and resource consumption across all workloads. These experiments confirmed the expected performance trends: replacing network communication with local calls



(a) Latency **(b)** Memory **(c)** CPU Utilization

Figure 6 Performance Characteristics for SCShowcase

Table 4 Performance Characteristics for RM* (microservice values represent the aggregate resource consumption of all distributed instances versus the single-process monolith)

Subject	Latency (ms)		Memory (GB)		CPU Utilization (%)	
	Microservice	Monolith	Microservice	Monolith	Microservice	Monolith
RM-10-0.2	180	**20**	2.03	**0.25**	34	**23**
RM-10-1.0	878	**587**	3.47	**1.79**	47	**31**
RM-100-0.02	1594	**281**	37.04	**4.94**	71	**52**

reduces both latency and resource consumption. Specifically, across all experiments, latency decreased by ≈65%, memory consumption by ≈72%, and CPU utilization by ≈33%. The observed significant reduction in memory and CPU usage is due to µjUniter's elimination of both network I/O and redundant Spring Cloud infrastructure. In the monolithic version, the system no longer requires multiple embedded servlet containers, service discovery clients, and the heavy reflection-based serialization overhead associated with REST-based inter-service communication.

5.4 RQ3: Automation Value

µjUniter automates the tedious and error-prone source code transformations required to transition from a distributed architecture to a centralized, modular monolith. To quantify this value and assess the degree of automation, we measured the lines of code removed, modified, and added using the CLOC (Count Lines of Code) tool [13], as well as the manual effort required to finalize the monolith. These results are presented in Table 5.

Table 5 µjUniter's CLOC Transformations and Automation Ratio

Subjects	**Added (CLOC)**	**Removed Infrastructure**	**$\mathcal{E}_{dist}$**	**Modified (CLOC)**	**Manual (CLOC)**	**Automation %**
Train Ticket	7520	2810	432	350	360	96.8%
Piggy Metrics	151	122	28	4	25	93.2%
SCShowcase	220	68	12	5	12	95.1%
RM-10-0.2	289	92	12	34	0	100%
RM-10-1.0	558	110	25	153	0	100%
RM-100-0.02	3250	915	164	487	0	100%

The Removed CLOC is categorized into two distinct groups: *Infrastructure* and $\mathcal{E}_{dist}$. Infrastructure removal targets Spring Cloud boilerplate, including Eureka/Consul service-discovery registrations, Zuul or Spring Cloud Gateway routing configurations, and `bootstrap.yml` files.

The $\mathcal{E}_{dist}$ category represents the pruning of distribution-induced dead code identified by our *Failure Model Refiner*, including `catch` blocks for `ConnectException`, `FeignException` handlers, and declarative resilience infrastructure such as `@CircuitBreaker` and `@HystrixCommand`.

We define the Automation % as the ratio of automatically transformed logic to the total lines of added, removed, and modified code in the final monolith. While µjUniter achieves 100% automation for the synthetic RM* subjects, third-party applications require a small margin of manual intervention, ranging from 3% to 7%. The 100% automation for RM* is enabled by our generator, which produces services with mutually exclusive property namespaces and standardized security headers, allowing the *Failure Model Refiner* and *Inconsistency Resolver* to prune all distribution-induced artifacts and wire all beans without encountering undecidable policy conflicts.

In contrast, the minor human-in-the-loop requirement for third-party subjects is not a fundamental limitation of µjUniter, but rather a necessity for resolving high-level architectural policies that are semantically undecidable via static analysis. Specifically, manual oversight was required to merge security policies, such as consolidating multiple `WebSecurityConfigurerAdapter` beans whose different microservices defined conflicting global CORS or CSRF settings. Similarly, property namespace conflicts necessitated manual resolution when identical keys in `application.properties` were used for different purposes across separate microservice domains. Table 6 provides the breakdown of the required manual effort.

Table 6 Manual Effort (CLOC / Time)

Subject	Security Merge	Property Namespace	Total (CLOC/h)
Train Ticket	210 / 2.5h	150 / 1.5h	360 / 4.0h
Piggy Metrics	16 / 0.3h	9 / 0.2h	25 / 0.5h
SCShowcase	8 / 0.3h	4 / 0.2h	12 / 0.5h
Total	**234 / 3.1h**	**163 / 1.9h**	**397 / 5.0h**

Despite these manual interventions, µjUniter automatically handled the migration's heavy lifting, including generating cross-module deep-copy utilities, redirecting REST clients to direct method invocations, and cleaning up signatures of checked exceptions. For a realistic third-party system like Train Ticket, the 96.8% automation rate implies that µjUniter successfully transformed over 11,000 lines of code, reducing the developer's task from a massive manual refactoring project to a focused configuration review.

6 Discussion

In this section, we discuss the applicability of our approach, with particular emphasis on the impact of preserving modularity and the limitations of our evaluation.

6.1 Applicability

Conceptually, Unification Refactoring is straightforward: extract the ASTs of distributed components, unify them into a single AST, identify the client-server endpoints, and replace network calls with local function calls, thereby producing the monolith. However, specific embodiments of Unification Refactoring would necessarily have to adhere to the conventions of the implementation ecosystem. For example, µjUniter's implementation is specifically tailored to the conventions and API of the Spring framework. Unification

REFACTORING should be applicable to other languages, libraries, or frameworks, but the required engineering effort can differ. Nevertheless, the main building blocks of UNIFICATION REFACTORING can be found in all major programming environments: AST manipulation and replacing network communication with local calls [15, 62]. Hence, we expect UNIFICATION REFACTORING to be widely applicable, but the availability of code analysis and transformation tools would affect the implementation effort.

6.2 Modularity and Ease of Reconciliation

The design of UNIFICATION REFACTORING provides significant additional benefits by preserving modularity during the unification process. When consolidated, originally distributed components remain isolated within individual modules. We observed that distributed deployment often masks configuration heterogeneity, such as disparate Java compiler versions, which only surfaces during consolidation. Because µjUniter maintains modular boundaries, we were able to exploit Java's backward compatibility: compiling each module with its original compiler while executing the unified monolith on a modern JVM. Thus, modular preservation simplifies the management of cross-version inconsistencies, which are inconsequential in isolation.

This structured environment is equally effective for resolving library version divergence. In a microservices architecture, independent teams may select different versions of shared libraries (e.g., JSON parsers or logging frameworks). When µjUniter detects such conflicts, it intentionally delegates resolution to the developer. We argue that maintaining multiple versions of the same library is frequently a symptom of postponed *technical debt* rather than a functional requirement.

Before performing an architectural transformation with µjUniter, developers should reconcile this debt. By requiring a manual update to a single, stable library version during the structural convergence phase, the developer ensures the resulting monolith is maintainable, secure, and performant. This approach avoids the complexity and overhead of runtime isolation hacks (like custom `ClassLoader`) while conforming to conventional, production-ready JVM deployment practices.

6.3 Threats to Validity and Limitations

Internal Validity: To ensure consistency and reliability of our evaluation across diverse microservice architectures, we performed targeted adjustments on our study subjects, which included the manual translation of three non-Java services in the Train Ticket benchmark and the resolution of legacy runtime errors in Piggy Metrics. While these adjustments modify the source, they were essential to ensure a uniform experimental baseline and broaden the architectural scope of our analysis. By stabilizing these subjects, we minimized the risk that runtime instability would influence our results.

External Validity: Our subject selection was guided by the objective of adhering to architectural relevance and modern development standards. Although the number of evaluation subjects is focused, this choice was necessitated by our rigorous inclusion criteria: we prioritized high-quality Spring Java exemplars over a larger number of outdated or unmaintained repositories. While this decision may influence the immediate generalizability of the results, it ensures that our findings are grounded in stable, contemporary microservice patterns.

Scale and Generalizability: We acknowledge that large-scale enterprise systems often exhibit higher complexity than open-source benchmarks. To bridge this gap, we introduced synthetic subjects designed to simulate larger architectural footprints. Although these

synthetic models were calibrated for available hardware resources, they provide a valuable controlled environment for observing system behavior at scale.

Scope of Metrics: Finally, this evaluation focuses on fundamental performance characteristics rather than environment-specific scalability or advanced deployment metrics. Given that such metrics are often highly dependent on specific infrastructure configurations, we focused on providing a robust baseline analysis. We view extending this work to specialized, deployment-specific performance profiling as a promising direction for future research.

7 Related Work

This work most closely relates to software architectural transformation and automated software refactoring for distribution, which we describe in turn.

Architectural Transformation

As business needs and resource constraints evolve, developers often need to transform the architecture of software systems. Hence, a rich body of prior work focuses on architectural transformations. Common approaches to architectural transformation focus on reconstructing or migrating the system structure [33, 69]. A recent state-of-the-art work investigates three frameworks and four case studies for building modular monoliths as an alternative or precursor to microservices [76]. Another recent work evaluates the performance and migration effort required to transform a monolith into microservices, creating a centralized service as an intermediate step [19]. IBM's Mono2Micro is a practical tool that decomposes monolithic Java applications into microservices by spatio-temporally clustering static and runtime data [39]. Micro2Micro optimizes the service boundaries of microservice systems and creates a centralized version as an intermediate step [12]. Work on service modularization, such as Service Cutter, analyzes coupling criteria to guide architectural slicing into well-defined services [31]. Google's Service Weaver provides a framework for writing modular monolith applications in Go that can be deployed either as a single process or as distributed microservices, preserving modularity while enabling runtime distribution [30], albeit requiring that applications be built from the ground up using a specific programming model. More recently, AI-driven approaches leverage LLMs to detect module boundaries and determine transformations that improve various characteristics of Java enterprise systems [84, 65].

These works share similar objectives—transforming architectures to optimize system performance. However, their transformations and tools focus on either increasing distribution by splitting monoliths into microservices or on optimizing existing microservice systems, rather than on reducing distribution. Although tools that create a centralized system version as an intermediate step are similar to µjUniter, we codify this architectural transformation as a new source-to-source code refactoring—Unification Refactoring—and demonstrate that it can be largely automated. To that end, this work has identified how to consolidate microservices into a modular monolith through automated refactoring, preserving modularity while eliminating network communication overhead. Unlike architectural reconstruction tools, µjUniter's pipeline merges microservice ASTs and rewires endpoints across modules. To our knowledge, this work is the first to identify the need for improved support for migrating microservices to a monolith to reduce distribution.

Automated Refactoring for Distribution

Several prior works present automated tools that assist with architectural refactoring in distributed systems. An automatic framework analyzes Java systems, reconstructs their conceptual architecture, and refactors them towards that architecture [73]. Another study critically reviews and categorizes several automatic refactoring strategies for decomposing monoliths into microservices [26]. Log2MS extracts execution logs to drive automated refactoring of monoliths into microservices, optimizing partitions based on runtime behavior [48]. Other automated refactoring tools, such as ARIES [6], apply rule-based source-to-source transformations to restructure legacy Java systems, while MicroDepGraph [70] builds dependency graphs from Java code and deployment descriptors to recommend microservice extractions. D-Goldilocks automatically rearchitects distributed systems to achieve an optimal degree of distribution [4]. As an intermediate step, D-Goldilocks also introduces a unification process, albeit specific to Node.js JavaScript applications, applied to running systems, and requiring programmer annotations.

These automated approaches also either focus on partitioning monoliths or redistributing distributed systems. None of them focuses on the problem of reducing distribution as an automated refactoring. Although we draw on prior automated refactoring solutions, our program analysis and transformation routines focus specifically on consolidating distributed microservices into a monolith.

8 Conclusion

This paper presents a novel refactoring—Unification Refactoring—that transforms a system comprising distributed components into a semantically equivalent centralized version, thereby closing the loop on architecture migration. Its source-to-source transformation pipeline merges abstract syntax trees and bridges control flow, consolidating distributed components into a centralized system. We reify Unification Refactoring as µjUniter, an automated refactoring framework that migrates Java Spring Boot microservices into a functionally equivalent modular monolith. Our evaluation of µjUniter shows that the resulting monolith maintains functionality while reducing latency and resource utilization. µjUniter's additional benefits include systematically refining failure models and pruning distribution-induced dead code, ensuring the removal of redundant network-handling logic in the resulting monolith. Overall, µjUniter provides substantial automation by eliminating the need to manually transform microservice source code and automatically handling over 96% of refactoring tasks. As technological advancements continue to impact software architecture, the need for automated architectural migrations will only grow. Unification Refactoring and µjUniter provide innovative designs and insights for this evolving field. As future work, we plan to extend µjUniter's applicability to other languages and frameworks. Another direction is to extend µjUniter's functionality to support the redistribution of microservices, with unification as an intermediate step. Because modern systems often need to both increase and decrease distribution to accommodate evolving requirements and constraints, Unification Refactoring fills the gap in support for architectural transformation.

References

1 Chaima Abid, Vahid Alizadeh, Marouane Kessentini, Thiago do Nascimento Ferreira, and Danny Dig. 30 years of software refactoring research: A systematic literature review. *arXiv preprint arXiv:2007.02194*, 2020.

2 Chaima Abid, Marouane Kessentini, Vahid Alizadeh, Mouna Dhaouadi, and Rick Kazman. How does refactoring impact security when improving quality? a security-aware refactoring approach. *IEEE Transactions on Software Engineering*, 48(3):864–878, 2020.

3 Omar Al-Debagy and Peter Martinek. A comparative review of microservices and monolithic architectures. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000149–000154. IEEE, 2018.

4 Kijin An and Eli Tilevich. D-goldilocks: Automatic redistribution of remote functionalities for performance and efficiency. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 251–260. IEEE, 2020.

5 Lujo Bauer, Andrew W Appel, and Edward W Felten. Mechanisms for secure modular programming in Java. *Software: Practice and Experience*, 33(5):461–480, 2003.

6 Gabriele Bavota, Andrea De Lucia, Andrian Marcus, Rocco Oliveto, and Fabio Palomba. Supporting extract class refactoring in eclipse: The aries project. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1419–1422. IEEE, 2012.

7 Ankur Bawiskar, Prashant Sawant, Vinayak Kankate, and BB Meshram. Spring framework: a companion to JavaEE. *IJCEM*, 1:41–49, 2012.

8 Thomas Betts. To microservices and back again – why segment went back to a Monolith, Apr 2020. URL: `https://www.infoq.com/news/2020/04/microservices-back-again/`.

9 Anshita Bhasin. Exploring Amazon Prime Video's architecture: Migrating from microservices to monolith for... Medium, May 2024. URL: `https://medium.com/@anshita.bhasin/exploring-amazon-prime-videos-architecture-migrating-from-microservices-to-monolith-for-aacbf9fabc73`.

10 Craig Box. Introducing istiod: simplifying the control plane. Technical report, Istio, 2020. URL: `https://istio.io/v1.5/blog/2020/istiod/`.

11 Tomas Cerny, Md Showkat Hossain Chy, Muhmmad Ashfakur Rahman Arju, Korn Sooksatra, Amr S Abdelfattah, and Valentina Lenarduzzi. A multi-variant benchmark for microservice systems in software engineering research. In *European Conference on Software Architecture*, pages 21–29. Springer, 2024.

12 Md Showkat Hossain Chy, Korn Sooksatra, Jorge Yero, and Tomas Cerny. Benchmarking Micro2Micro transformation: an approach with GNN and VAE. *Cluster Computing*, 27(4):4171–4185, 2024.

13 Al Danial. Cloc: Count lines of code, 2017. URL: `https://cloc.sourceforge.net/`.

14 Paul Deitel. Understanding Java 9 modules. Oracle, 2017. URL: `https://www.oracle.com/corporate/features/understanding-java-9-modules.html`.

15 DevZery. Asts meaning: A complete programming guide. `https://www.devzery.com/post/asts-meaning-a-complete-guide-to-abstract-syntax-trees-in-programming`, 2025.

16 Edsger Wybe Dijkstra. Structured programming. 1970.

17 Oliver Drotbohm. Spring blog. Introducing Spring Modulith, Oct 2022. URL: `https://spring.io/blog/2022/10/21/introducing-spring-modulith`.

18 Hassan Farsi, Driss Allaki, Abdeslam En-Nouaary, and Mohamed Dahchour. A graph-based solution to deal with cyclic dependencies in microservices architecture. In *2022 9th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 254–259. IEEE, 2022.

19 Diogo Faustino, Nuno Gonçalves, Manuel Portela, and António Rito Silva. Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation. *Performance Evaluation*, 164:102411, 2024.

20 Stefan Fischer, Pirmin Urbanke, Rudolf Ramler, Monika Steidl, and Michael Felderer. An overview of microservice-based systems used for evaluation in testing and monitoring: A systematic mapping study. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, pages 182–192, 2024.

21 The Apache Software Foundation. Apache JMeter™, 1998. URL: `https://jmeter.apache.org/`.

22 The Apache Software Foundation. Apache Maven project. https://maven.apache.org/what-is-maven.html, 2004.

23 The Apache Software Foundation. Guide to working with multiple modules, 2004. URL: https://maven.apache.org/guides/mini/guide-multiple-modules.html.

24 Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

25 Jonas Fritzsch, Justus Bogner, Stefan Wagner, and Alfred Zimmermann. Microservices migration in industry: intentions, strategies, and challenges. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 481–490. IEEE, 2019.

26 Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. From monolith to microservices: A classification of refactoring approaches. In *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 128–141. Springer, 2018.

27 FudanSELab. Train ticket a benchmark microservice system, August 2022. URL: https://github.com/FudanSELab/train-ticket.

28 John Fx. Serverless was a big mistake... says amazon. YouTube, May 2023. URL: https://www.youtube.com/watch?v=qQk94CjRvIs.

29 Gatling. Gatling documentation. https://docs.gatling.io/, 2012.

30 Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker, Parveen Patel, Ivan Posva, and Amin Vahdat. Towards modern development of cloud applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 110–117, 2023.

31 Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. Service cutter: A systematic approach to service decomposition. In *European Conference on Service-Oriented and Cloud Computing*, pages 185–200. Springer, 2016.

32 Muhammad Hafiz Hasan, Mohd Hafeez Osman, Novia Indriaty Admodisastro, and Muhamad Sufri Muhammad. A quality driven framework for decomposing legacy monolith applications to microservice architecture. Research Square, 2023. doi:10.21203/rs.3.rs-3060410/v1.

33 Wilhelm Hasselbring. Software architecture: Past, present, future. In *The essence of software engineering*, pages 169–184. Springer International Publishing Cham, 2018.

34 Amy E Hodler and Mark Needham. Graph data science using neo4j. In *Massive Graph Analytics*, pages 433–457. Chapman and Hall/CRC, 2022.

35 Scott M. Fulton III. Amazon prime video's microservices move doesn't lead to a monolith after all. The New Stack, Jun 2023. URL: https://thenewstack.io/amazon-prime-videos-microservices-move-doesnt-lead-to-a-monolith-after-all/.

36 ISO/IEC/IEEE. Software engineering — software life cycle processes — maintenance, 2022. doi:10.1109/IEEESTD.2022.9690131.

37 JetBrains. Jetbrains Microservices. https://www.jetbrains.com/lp/devecosystem-2022/microservices/, 2023.

38 Xinyang Jia. The role and importance of software testing in software quality management. *Journal of Industry and Engineering Management*, 1(4):39–44, 2023.

39 Anup K Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. Mono2micro: a practical and effective tool for decomposing monolithic Java applications to microservices. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1214–1224, 2021.

40 Raffi Khatchadourian, Jason Sawin, and Atanas Rountev. Automated refactoring of legacy Java software to enumerated types. In *2007 IEEE International Conference on Software Maintenance*, pages 224–233. IEEE, 2007.

41 Marcin Kolny. Scaling up the prime video audio/video monitoring service and reducing costs by 90%, Mar 2023. URL: https://www.wudsn.com/productions/www/site/news/2023/2023-05-08-microservices-01.pdf.

42 Koushik Kothagal. *Modular Programming in Java 9: Build large scale applications using Java modularity and Project Jigsaw*. Packt Publishing Ltd, 2017.

43 Anthony Kwan, Hans-Arno Jacobsen, Allen Chan, and Suzette Samoojh. Microservices in the modern software world. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, pages 297–299, 2016.

44 Young-Woo Kwon. Automated s/w reengineering for fault-tolerant and energy-efficient distributed execution. In *2013 IEEE International Conference on Software Maintenance*, pages 582–585. IEEE, 2013.

45 Young-Woo Kwon and Eli Tilevich. Cloud refactoring: automated transitioning to cloud-based services. *Automated Software Engineering*, 21(3):345–372, 2014.

46 Mahesh Lal. *Neo4j graph data modeling*. Packt Publishing Ltd, 2015.

47 Duc Minh Le, Pooyan Behnamghader, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. An empirical study of architectural change in open-source software systems. In *2015 IEEE/ACM 12th working conference on mining software repositories*, pages 235–245. IEEE, 2015.

48 Bo Liu, Jingliu Xiong, Qiurong Ren, Shmuel Tyszberowicz, and Zheng Yang. Log2ms: a framework for automated refactoring monolith into microservices using execution logs. In *2022 IEEE International Conference on Web Services (ICWS)*, pages 391–396. IEEE, 2022.

49 Yin Liu, Kijin An, and Eli Tilevich. Rt-trust: automated refactoring for trusted execution under real-time constraints. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 175–187, 2018.

50 Alexander Lukyanchikov. Piggy metrics, July 2018. URL: `https://github.com/sqshq/piggymetrics`.

51 Nabor C Mendonça, Craig Box, Costin Manolache, and Louis Ryan. The monolith strikes back: Why istio migrated from microservices to a monolithic architecture. *IEEE software*, 38(5):17–22, 2021.

52 Naftaly H Minsky and Victoria Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(3):273–305, 2000.

53 Mayank Mishra, Shruti Kunde, and Manoj Nambiar. Cracking the monolith: Challenges in data transitioning to cloud native architectures. In *Proceedings of the 12th European conference on software architecture: companion proceedings*, pages 1–4, 2018.

54 Piotr Mińkowski. Microservices with Spring cloud advanced demo project, May 2024. URL: `https://github.com/piomin/sample-spring-microservices-new/tree/master`.

55 Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. Fault-tolerant distributed reactive programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, pages 1–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018.

56 Rodrigo Morales, Rubén Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. Earmo: An energy-aware refactoring approach for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering*, pages 59–59, 2018.

57 Conor Muldoon, Levent Görgü, John J O'Sullivan, Wim G Meijer, Bartholomew Masterson, and Gregory MP O'Hare. Engineering testable and maintainable software with Spring Boot and React. *Authorea Preprints*, 2023.

58 Kshirasagar Naik and Priyadarshi Tripathy. *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011.

59 Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, 2005.

60 Vikram Nitin, Shubhi Asthana, Baishakhi Ray, and Rahul Krishna. Cargo: AI-guided dependency analysis for migrating monolithic applications to microservices architecture.

In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

61 Rory V O'Connor, Peter Elger, and Paul M Clarke. Continuous software engineering—a microservices architecture perspective. *Journal of Software: Evolution and Process*, 29(11):e1866, 2017.

62 Knowledgelark Encyclopedia of Daily Life. Handling network request mocks in tests. `https://www.knowledgelark.com/archives/5263`, 2025.

63 William F Opdyke. *Refactoring object-oriented frameworks*. University of Illinois at Urbana-Champaign, 1992.

64 Amey Padvekar and Vikaskumar Gupta. Comparative analysis of monolithic vs. distributed architecture. *International Journal of Advanced Research in Science, Communication and Technology*, pages 433–442, 06 2024. `doi:10.48175/IJARSCT-18946`.

65 Indranil Palit and Tushar Sharma. Generating refactored code accurately using reinforcement learning. *arXiv preprint arXiv:2412.18035*, 2024.

66 Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of Java source code. *Software: Practice and Experience*, 46(9):1155–1179, 2016.

67 Perforce, Inc. 2023 Java developer productivity report. `https://www.jrebel.com/resources/java-developer-productivity-report-2023`, 2023.

68 Evaggelia Pitoura and Bharat Bhargava. Data consistency in intermittently connected distributed systems. *IEEE Transactions on knowledge and data engineering*, 11(6):896–915, 2002.

69 Konstantinos Plakidas, Daniel Schall, and Uwe Zdun. Software migration and architecture evolution with industrial platforms: A multi-case study. In *European Conference on Software Architecture*, pages 336–343. Springer, 2018.

70 Mohammad Imranur Rahman, Sebastiano Panichella, and Davide Taibi. A curated dataset of microservices-based systems. In *Joint of the Summer School on Software Maintenance and Evolution*. CEUR-WS, 2019.

71 Mark Russinovich. Microservices: An application revolution powered by the cloud, May 2023. URL: `https://azure.microsoft.com/en-us/blog/microservices-an-application-revolution-powered-by-the-cloud/`.

72 Max Schäfer, Torbjörn Ekman, and Oege De Moor. Sound and extensible renaming for Java. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 277–294, 2008.

73 Frederik Schmidt, Stephen G MacDonell, and Andrew M Connor. An automatic architecture reconstruction and refactoring framework. In *Software engineering research, management and applications 2011*, pages 95–111. Springer, 2012.

74 Sheldon Smith, Ethan Robinson, Timmy Frederiksen, Trae Stevens, Tomas Cerny, Miroslav Bures, and Davide Taibi. Benchmarks for end-to-end microservices testing. In *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 60–66. IEEE, 2023.

75 Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.

76 Ruoyu Su and Xiaozhou Li. Modular monolith: Is this the trend in software architecture? In *Proceedings of the 1st International Workshop on New Trends in Software Architecture*, pages 10–13, 2024.

77 Kridanto Surendro, Wikan Danar Sunindyo, et al. Circuit breaker in microservices: State of the art and future prospects. In *IOP Conference Series: Materials Science and Engineering*, volume 1077, page 012065. IOP Publishing, 2021.

78 Wesley Tansey and Eli Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 295–312, 2008.

79 Wesley Tansey and Eli Tilevich. Efficient automated marshaling of C++ data structures for MPI applications. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, 2008. doi:10.1109/IPDPS.2008.4536307.

80 Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Automatic Java application partitioning. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming*, pages 178–204, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

81 Victor Velepucha and Pamela Flores. A survey on microservices architecture: Principles, patterns and migration challenges. *IEEE access*, 11:88339–88358, 2023.

82 Andrew Walker, Ian Laird, and Tomas Cerny. On automatic software architecture reconstruction of microservice applications. In *Information Science and Applications: Proceedings of ICISA 2020*, pages 223–234. Springer, 2021.

83 W Eric Wong, Joseph R Horgan, Saul London, and Hiralal Agrawal. A study of effective regression testing in practice. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pages 264–274. IEEE, 1997.

84 Yisen Xu, Feng Lin, Jinqiu Yang, Nikolaos Tsantalis, et al. Mantra: Enhancing automated method-level refactoring with contextual rag and multi-agent llm collaboration. *arXiv preprint arXiv:2503.14340*, 2025.

85 Yung-Yu Zhuang, Wei Kuo, and Shang-Chun Tseng. Resolving the Java representation exposure problem with an AST-based deep copy and flexible alias ownership system. *Electronics*, 13(2):350, 2024.

86 Mohamed Zouari and Ismael Bouassida Rodriguez. Towards automated deployment of distributed adaptation systems. In *European Conference on Software Architecture*, pages 336–339. Springer, 2013.