TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Karl-Erik Hein 221339IAPM

Abstract Syntax Tree-Based Tooling For Java Framework Migration

Master's Thesis

Supervisor: Gert Kanter
PhD

Tallinn 2025

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Karl-Erik Hein 221339IAPM

Abstraktsel süntaksipuul põhinev tööriist Java raamistike migreerimiseks

Magistritöö

Juhendaja: Gert Kanter
PhD

Tallinn 2025

Author’s Declaration of Originality

I hereby certify that I am the sole author of this thesis and that this thesis has not been presented for examination or submitted for defense anywhere else. All used materials, references to the literature, and work of others have been cited.

Author: Karl-Erik Hein

20.05.2025

Abstract

Modernizing enterprise services often involves legacy framework migrations - like moving an older Dropwizard-based system to a newer and more comprehensive Spring Boot. This thesis evaluates the effective automation of such migrations through AST-based refactoring using OpenRewrite, with an emphasis on designing modular, production-ready migration recipes. To benchmark the AST-driven approach, we compare it against an LLM-assisted refactoring (using OpenAI's *o3-mini* model) and traditional manual rewriting. Two case studies underpin the evaluation: one moderately complex open-source service and one proprietary closed-source system, each migrated from Dropwizard 1.3 to Spring Boot 2.7.x. Our assessment examines the time required and the extent of automation achieved, while ensuring functional correctness through automated test suites and manual verification.

The results indicate that AST-based methods yield faster and more comprehensive conversions but require significant up-front effort to develop robust recipes. LLM-based refactoring accelerates simpler tasks yet shows inconsistent reliability, particularly with proprietary code, often requiring iterative developer guidance. Manual migration remains the most reliable approach but is also the most time-intensive.

We observed common technical hurdles across the automated approaches, including adapting types and annotations, managing dependencies and configuration properties, and automating complex syntax changes. *Key contributions* of this work include a detailed methodology for crafting AST-based migration recipes, an empirical comparison of three distinct migration strategies, and the release of the resulting recipe framework as an open-source resource for the community. Overall, the findings suggest that a blended strategy—combining AST automation for core refactoring tasks, selective LLM assistance, and careful manual validation—may provide the most effective balance of efficiency and reliability in such framework migrations.

The thesis is in English and contains 66 pages of text, 7 chapters, 20 figures, 5 tables.

Annotatsioon

Abstraktsel süntaksipuul põhinev tööriist Java raamistike migreerimiseks

Tarkvara kaasajastamisel tuleb sageli teha raamistikuvahelist üleviimist, nagu näiteks Dropwizardi raamistikust ulatuslikumasse Spring Booti. Käesolev töö käsitleb, kuidas seda protsessi automatiseerida, rakendades abstraktse süntaksipuu põhist refaktoriseerimist OpenRewrite'iga ning kujundades tootmiskõlblikke retsepte. Lähenemisviisi võrreldakse nii suurte keelemudelite (*OpenAI o3-mini*) toel tehtud kui ka täielikult käsitsi läbiviidud migreerimisega. Analüüs rajaneb kahel uuringul: ühel keskmise keerukusega avatud lähtekoodiga teenusel ja ühel suletud lähtekoodiga rakendusel. Hinnanguprotsessis arvestati ajakulu, automatiseerituse määra ja funktsionaalset vastavust, mida kontrolliti nii automaatsete testidega kui ka käsitsi valideerimisega.

Tulemused osutavad, et kuigi abstraktse süntaksipuu põhine meetod võimaldab kiireimat raamistikuvahetust, nõuab see põhjalikku retseptide ettevalmistust. Suur keelemudel kiirendab lihtsamaid ümberkirjutamisetappe, kuid osutub ebaühtlaseks, jäädes hätta suuremate klasside, puuduvate detailide ja privaatse koodiga, mistõttu vajab see märkimisväärset käsitsi juhendamist ja parandamist. Mõlema automatiseeritud lähenemise puhul olid peamisteks raskusteks tüüpide ja annotatsioonide kohandamine, sõltuvuste ja konfiguratsioonifailide haldamine ning keerukate süntaksistruktuuride automaatne ümberkirjutamine. Kokkuvõtvalt võib öelda, et AST-põhine meetod oli kõige efektiivsem, pakkudes parimat tasakaalu kiiruse ja töökindluse vahel, ent vajas siiski käsitsi viimistlemist, eriti keerukamate loogikaosade ja testide kohandamisel. Loodud siirderetseptid on avalikustatud avatud lähtekoodina, et soodustada kogukonna laiemat kasutust ja panust.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 66 leheküljel, 7 peatükki, 20 joonist, 5 tabelit.

List of Abbreviations and Terms

API	*Application Programming Interface* - allows communication between different software.
AST	*Abstract Syntax Tree* – A tree representation of the abstract syntactic structure of source code.
DAO	*Data Access Object* – A design pattern that provides an abstract interface to a database or other persistence mechanism, decoupling the data access logic from business logic.
Dropwizard	A Java framework for building RESTful web services.
DSL	*Domain Specific Language* – is a computer language specialized to a particular application domain.
Framework	A collection of pre-built code that provides structure for application development.
IoC	*Inversion of Control* – A design pattern where frameworks manage object dependencies.
Kubernetes (*K8s*)	An open-source platform for automating the deployment, scaling, and management of containerized applications.
Library	A collection of reusable code that developers can include in their projects.
LLM	*Large language model* - A machine learning model trained on large text datasets for language tasks.
LST	*Lossless Semantic Tree* – A modified version of Abstract Syntax tree, which preserves syntactic details of the code.
Native Images	Compiled application images that run directly as machine code.
OpenRewrite	An open-source framework to change code based on recipes.
ORM	*Object-Relational Mapping* - A programming technique that allows database access via objects instead of writing queries.
RESTful	An architectural style for designing web services using HTTP.
Spring	A framework for building Java enterprise applications.
Spring Boot	A Spring framework extension for easier, standalone Java applications.

Table of Contents

List of Figures

List of Tables

1 Introduction

We consistently face significant challenges in maintaining legacy systems—critical infrastructures that grow increasingly brittle, costly, and insecure with age [1]. Despite being fundamental to many business operations, these systems are often built on outdated technologies that demand substantial maintenance. Their rigidity impedes integration with modern platforms, resulting in technological stagnation. Nonetheless, legacy systems remain valuable because they encapsulate years of organizational knowledge and investment.

Migrating an application within the same framework already carry considerable cost and complexity [2], and rewriting systems from scratch can be riskier and more expensive than anticipated [3]. Manual migration methods are especially time-intensive, prone to human error, and poorly suited for large-scale transitions. Tools that leverage *Abstract Syntax Trees* for structured code refactoring [4, 5] have emerged to assist in automating these migrations by enabling precise code analysis and transformation, thus reducing the likelihood of human error. At the same time, *Large Language Models* offer a complementary approach by adaptively interpreting and generating code, potentially facilitating translation across different frameworks with varying degrees of success [6, 7].

However, while AST-based and LLM-driven techniques show promise for cross-framework Java migrations, their real-world applicability remains underexplored—particularly in transitions between frameworks such as Dropwizard [8, 9] and Spring Boot [10, 11]. Dropwizard, a popular Java framework, is increasingly overshadowed by Spring Boot's greater flexibility, richer feature set, and more active community support. In an anonymized company, most of the technology stack is built on Spring Boot, yet numerous older services continue to run on Dropwizard. For stack unification and system longevity, the organization seeks to migrate these services from Dropwizard to Spring Boot without losing accumulated business logic. Such a migration is a labor-intensive and error-prone process if done entirely by hand.

Industry experience suggests that automating repetitive migration steps can significantly enhance developer productivity and reduce modernization costs [12, 13]. Accordingly, this thesis examines how advanced tooling—ranging from AST-based refactoring techniques to LLM-assisted code generation—can streamline the Dropwizard to Spring Boot migration process by minimizing manual effort and improving reliability. Our goal is to bridge this methodological gap and deliver both insights and practical tools to support more effective modernization of legacy systems.

1.1 Research Questions

This investigation is guided by the following research questions:

RQ1: How should we design and structure a *proof-of-concept migration recipe* to address the specific challenges of cross-framework transitions, with minimal developer intervention?

RQ2: How extensively can we automate the migration from Dropwizard to Spring Boot using AST-based tools, ensuring minimal changes and preserving functional parity?

RQ3: Where do AST-based tools like OpenRewrite fit on the spectrum between fully manual migrations and LLM-driven transformations, and what are the trade-offs between these approaches?

By addressing these questions, we aim to provide valuable insights into reducing both the cost and complexity of Java framework migrations. In doing so, we aspire to contribute not only to academic research on automated refactoring, but also to offer guidance for industry practitioners seeking to modernize their software stacks more efficiently.

1.1.1 Contributions

This thesis makes three principal contributions. First, it proposes a modular, AST-based recipe methodology that (partially) automates the migration of Dropwizard services to Spring Boot while retaining existing business logic. Second, it presents an empirical comparison of three migration strategies—fully manual, LLM-assisted, and AST-driven—across

two different case studies. Third, it releases the resulting recipe framework as an open-source library[1], thereby enabling other practitioners to replicate and extend our work.

1.2 Research Design

To address our research questions, we conduct two complementary case studies that reflect distinct migration scenarios. The first focuses on a publicly accessible, moderately complex Dropwizard application, whereas the second examines a proprietary service that embeds specialised business logic and bespoke integrations. In both studies we migrate the code base from Dropwizard 1.3 to Spring Boot 2.7.x.

Each migration proceeds in two stages. We begin with a purely *manual* port to establish a baseline for effort and correctness; this baseline is then compared against two automated approaches—AST-driven refactoring with OpenRewrite and LLM-assisted refactoring with *o3-mini*. Comparing the three pathways highlights the unique challenges and benefits of each strategy under realistic conditions.

Evaluation relies on explicit quantitative indicators defined in Section 4.1. The *Automation Completeness Score* measures how much of the migration is achieved automatically, considering line changes, dependency updates, and structural refactorings. The *Time Factor* records the speed-up of each automated approach relative to the manual baseline. The *Total Score* combines the previous two, presenting an overall score.

Finally, *functional parity* is ensured through automated tests in addition to manual validation — ensuring that the migrated service functions equivalently to its Dropwizard predecessor.

Taken together, these case studies and metrics enable a *rigorous, quantitative* comparison of AST-based and LLM-assisted migration techniques, clarifying both the degree of automation each approach affords and the contexts in which each one proves most effective.

[1]Dropwizard to Spring Boot Recipe Repository, Accessed: 2025-04-13, URL: `https://github.com/Fossur/openrewrite-dropwizard-springboot-recipe`

2 Background

Legacy software systems underpin critical business operations but increasingly pose significant challenges due to aging technologies, accumulated technical debt, and evolving business requirements [1, 14]. The scale of this issue is reflected in public sector spending; for example, in 2019, the U.S. government allocated over $90 billion to IT, nearly 80% of which was dedicated to maintaining legacy systems [15].

Organizations grappling with legacy systems face multiple interconnected challenges, including high maintenance costs due to obsolete technology and a scarcity of skilled personnel, security vulnerabilities stemming from outdated protective measures, and integration difficulties with modern platforms [1]. Additionally, maintaining legacy systems becomes increasingly complex as knowledge gaps widen when original developers retire or move on [16].

To address these challenges, several modernization strategies exist. Basic migration methods include the *Lift-and-Shift* approach, which relocates applications to cloud platforms without altering their original architectural design [17]. More comprehensive strategies outlined by Marchezan et al. include *Replace* (adopting off-the-shelf solutions), *Maintain* (minimal updates), *Evolve* (incremental improvements), *Re-engineer* (systematic refactoring), and *Migrate* (transition to modern technology stacks) [1].

For organizations whose legacy systems retain high strategic value, migration and re-engineering often emerge as the most viable approaches [1, 3]. Transitioning to microservice architectures is a notable example of these modernization strategies [18, 19, 20]. While rewriting legacy systems from scratch might initially seem appealing for providing a clean slate, this approach frequently incurs higher costs, longer timelines, and potential risks including the loss of embedded business logic and institutional knowledge [3]. Therefore, migration strategies typically offer a more balanced solution, preserving existing business functions and minimizing disruption [15, 16].

While manual migration has been traditionally prevalent, it is often found to be inconsistent, complex, and expensive in both time and resources [3]. Danske Bank's recent large-scale modernization, for instance, involving thousands of servers and workloads, was estimated to take five to eight years if performed manually [21]. By integrating automation, they successfully cut program costs and timelines by *50%* [21]. Consequently, organizations are increasingly exploring automated migration approaches to effectively mitigate these challenges.

2.1 Automated Code Transformation Tools

When undertaking large-scale software refactoring, automated code transformation tools can dramatically streamline the process. These tools programmatically parse, modify, and rewrite code, reducing manual effort, ensuring consistency, and mitigating risks inherent in migration projects. Below in Table 1, we compare five prominent tools with a focus on their suitability for Java-based migration from Dropwizard to Spring Boot.

Criteria	**OpenRewrite**	**Spoon**	**Refaster**	**Codemod**	**jscodeshift**
Supports Java	✓	✓	✓	×	×
Large existing recipe-base	✓	×	×	✓	✓
Handles complex refactorings	✓	✓	×	×	×
Widespread popularity	✓	×	×	✓	✓

Table 1. Comparison of code transformation tools

2.1.1 Codemod & jscodeshift

Codemod [22] was initially developed for Python codebase modifications using regular expressions. Its JavaScript counterpart, *jscodeshift* [23], provides AST-based transformations specifically tailored for JavaScript. However, neither Codemod nor jscodeshift supports Java directly or offers ready-to-use recipes for Java framework migrations. Therefore, these tools are unsuitable for our specific migration needs.

2.1.2 Refaster

Refaster [24, 25, 26] extends Google's Error Prone framework with template-based code transformations. While useful for targeted, small-scale refactorings, Refaster lacks the flexibility required for extensive architectural changes, making it insufficient for our complex Dropwizard to Spring Boot migration. In addition, OpenRewrite already supports Refaster style recipes out-of-the-box [27], making it unnecessary to use this tool.

2.1.3 Spoon

Spoon [4] enables fine-grained AST manipulation for Java, providing robust control over transformations at the class, method, and expression levels. Although Spoon is powerful, it offers minimal pre-built, community-made recipes, requiring significant upfront development effort. Consequently, while viable, Spoon would demand substantial custom implementation for handling the nuances of our framework transition.

2.1.4 OpenRewrite: The Selected Tool

For this project, we selected *OpenRewrite* [28] to automate the Dropwizard to Spring Boot migration because it offered key advantages over other tools. It uses a *Lossless Semantic Tree* [29], which preserves type changes and formatting information – allowing for complex refactorings. OpenRewrite provides many ready-made recipes for common Java tasks and upgrades [30], which saves considerable development time. The tool benefits from a strong, active community that provides helpful support and documentation. Taken together, OpenRewrite's precise LST, extensive recipe library, and strong community support made it the most promising foundation for the AST-based migration approach.

2.2 Leveraging Large Language Models for Code Migration

Alongside automated code transformation tools, we independently explore Large Language Models due to their distinctive ability to understand nuanced programming contexts, generate flexible transformations, and handle ambiguous migration scenarios. In this section, we evaluate how LLMs can complement existing tools, acknowledge their limitations, and propose effective strategies for their usage in migrating our Java-based applications from Dropwizard to Spring Boot.

2.2.1 Developer Usage of LLMs

The adoption of tools such as ChatGPT [31] and GitHub Copilot [32] has significantly impacted software engineering by enabling intuitive natural language interactions and intelligent code generation. Almeida et al. [6] demonstrated the potential of LLMs in library migration tasks, highlighting their usefulness but also noting that developers must still manage issues like missing imports.

Barke et al. [33] identified two primary ways developers utilize LLMs:

1. **Acceleration mode:** Employing LLMs for straightforward tasks, such as generating boilerplate code, significantly speeding up repetitive migration efforts.
2. **Exploration mode:** Using LLMs as interactive collaborators to navigate complex, uncertain transitions, particularly useful in migration scenarios involving new design patterns and frameworks.

In this thesis, our primary focus will be on leveraging the *exploration mode*, as it aligns closely with the uncertainties and complexities inherent in migrating between two distinct Java frameworks.

2.2.2 Challenges and Limitations

LLMs introduce several challenges, most notably hallucinations and biases. Liu et al. [34] identified three types of hallucinations that can impact code quality:

1. **Intent Conflicting:** Generated code deviates from intended functionality.
2. **Context Deviation:** Inconsistent, unnecessarily repetitive and dead code.
3. **Knowledge Conflicting:** Incorrectly utilizing variables or invalid API usage.

Additionally, Huang et al. [35] warned about biases embedded in LLM-generated code, which could propagate discriminatory or unfair outcomes, particularly within decision-critical systems.

2.2.3 Effective Usage Strategies

Success with LLMs relies heavily on effective prompt engineering. Almeida et al. [6] outlined three key prompting techniques:

1. **Zero-Shot Prompting:** Direct task instructions, effective for simple or well-known migration patterns.
2. **One-Shot Prompting:** Providing specific examples to guide transformations, increasing the accuracy of LLM outputs.
3. **Chain-of-Thought Prompting:** Structuring prompts step-by-step, particularly effective in reducing logical and functional errors.

2.2.4 Security and Privacy Considerations

Recent incidents, such as Samsung's data breach through ChatGPT usage [36], underscore critical security and privacy concerns. Organizations must enforce rigorous data management protocols and provide training to mitigate risks, particularly when using LLMs with proprietary code [37].

2.2.5 Model Selection

When selecting an appropriate LLM for our migration tasks, we considered various benchmarks and leaderboards, such as those provided by Aider [38] and the LiveBench benchmark by White et al. [39]. Based on these evaluations, we chose OpenAI's o3-mini model with high reasoning [40] due to its very good performance in reasoning-intensive coding tasks, directly addressing the complexities involved in our framework migration scenario.

2.2.6 Conclusion

While LLMs offer powerful capabilities for code migration, their effective use requires balancing automation benefits against potential risks. Success depends on understanding their limitations, employing strategic prompt engineering, and maintaining strong security practices. As these tools evolve, continued research into best practices and robust governance frameworks will be essential for their successful integration into software engineering workflows.

By utilizing the *exploration mode* and *zero-shot prompting*, we specifically aim to effectively navigate the uncertainty and complexity encountered in real-world migration contexts, where predefined examples are rarely available. The choice of the o3-mini model further

supports our strategy, given its proven strengths in managing nuanced programming challenges.

2.3 Comparative Analysis of Dropwizard and Spring Boot

To provide context for our migration, we briefly analyze *Dropwizard* [8] and *Spring Boot* [41]. Both frameworks facilitate developing standalone, production-ready Java web applications, but they differ significantly in philosophy and capabilities. Understanding these differences clarifies our motivation to migrate from Dropwizard to Spring Boot.

2.3.1 Design and Configuration Philosophies

Spring Boot emphasizes developer productivity via *convention over configuration*. It leverages autoconfiguration, reducing setup effort [41], and relies on Spring Framework's IoC container for managing dependencies [42]. Its layered configuration approach (YAML, properties files, environment variables, etc.) offers flexibility across deployment scenarios [41]. Additionally, type-safe binding with `@ConfigurationProperties` enhances robustness.

Dropwizard, in contrast, favors explicitness and minimalism [8]. It integrates mature libraries (Jetty, Jersey, Jackson) but requires explicit component wiring, usually within an `Application` subclass [8, 43]. Its simpler YAML configuration mapped directly to Java classes ensures clarity but provides limited flexibility compared to Spring Boot [8, 43].

2.3.2 General capabilities

A primary driver for migration involves each framework's broader capabilities, especially in modern architectures. *Spring Boot* offers comprehensive support through *starters* [41], integrating seamlessly with many Spring projects and third-party libraries. For microservice scenarios, *Spring Cloud* provides solutions for service discovery, load balancing, distributed tracing, and more [44], bolstered by production-ready *Actuator* endpoints and GraalVM native image support [45]. Although Spring Boot's JVM startup may be slower due to autoconfiguration [46, 43], its runtime performance is comparable, and native compilation significantly improves startup times [45].

Dropwizard focuses on core RESTful services [8], offering fewer out-of-the-box integrations for complex microservice patterns [43, 8]. Implementing advanced features like service discovery and circuit-breaking frequently relies on external platforms or custom extensions. Its main performance advantage is a faster JVM startup and lower footprint [46, 43], but the lack of native compilation support hampers certain modern deployment use cases.

2.3.3 Support and Community

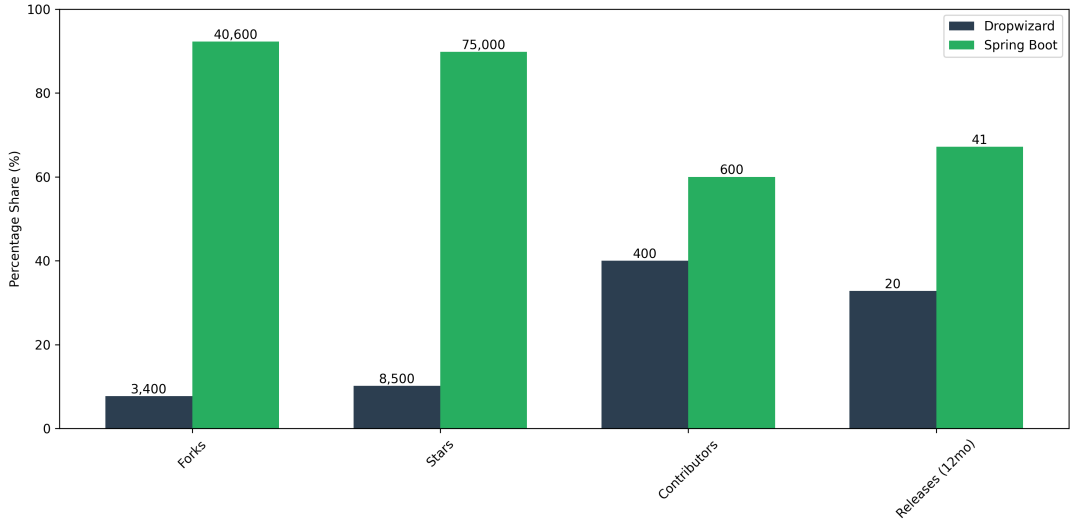

Figure 1. Relative comparison of Dropwizard vs Spring Boot GitHub metrics [9, 11]

Spring Boot benefits from VMware's corporate backing, commercial support options, and a thriving global user base [47, 10]. As seen in Figure 1, it also shows significantly higher GitHub metrics (forks, stars, contributors, and release frequency) than *Dropwizard*, reflecting frequent releases, extensive documentation, and an active developer community that helps ensure long-term viability.

Dropwizard, by contrast, is community-maintained without corporate sponsorship. It has a smaller yet dedicated user community [9, 8], which limit available resources and extensions compared to the far broader Spring ecosystem.

2.3.4 Conclusion

Table 2 provides our summarized view of the key differences influencing the decision to migrate.

Feature	**Spring Boot**	**Dropwizard**
Philosophy	Convention over config, Auto-configuration	Explicit config, Curated libraries
Ecosystem	*Vast & Integrated* (Spring projects, Starters)	Focused & Limited (Core libs, Bundles)
Cloud-Native Features	*Strong* (Config flexibility, Native Images, K8s alignment)	Capable, less integrated
Extensibility	High via starters & auto-config	Moderate via bundles & manual integration
Support Model	Commercial option, *large community*	(small) community-driven
Learning Curve	Easy start, potentially complex mastery	Steeper start (explicit), simpler scope
Performance (JVM)	Slower startup, higher footprint	Faster startup, lower footprint
Performance (Native)	*Supported* (fast startup, low footprint)	Not natively supported

Table 2. Key differences influencing migration from Dropwizard to Spring Boot

Spring Boot offers a more comprehensive set of tools that better support modern software development, especially for cloud-based applications. The framework provides a broader ecosystem, stronger community support, and more flexible features. While Dropwizard has its merits, Spring Boot's extensive capabilities and industry-wide adoption make it a more attractive choice for long-term development needs.

3 Migration

3.1 Overall Approach

This section details our strategy for migrating legacy Dropwizard services to Spring Boot. The primary drivers for this undertaking are reducing technical debt, enhancing maintainability, and improving integration with our modern infrastructure stack.

3.1.1 Framework Version Selection

We selected Spring Boot version 2.7.18 for this migration due to its compatibility with Java 8 [48]. Given Java 8's continued prevalence in enterprise environments [49], this choice avoids the immediate need to upgrade the Java version for older services, which would introduce significant overhead to the migration effort. Spring Boot versions beyond 2.7.x require Java 17 or higher [48].

3.1.2 Git Branch Management Strategy

To manage and analyze the migration process effectively, each migration task (corresponding to a specific service and approach – manual, LLM, or AST) will be performed in dedicated Git branches, originating from the baseline commit of the original service. Each migration will proceed in two main stages, captured distinctly within the branch structure to isolate effort. The *Initial Stage* represents the code immediately after applying the primary migration method (e.g., the raw output from LLM/AST tooling, or the initial manual porting effort). Subsequently, manual fixes and adjustments needed to meet the validation criteria constitute the *Finalized Stage*. This work will be performed on a separate branch forked from the initial state. This two-stage branching facilitates a clear comparison of the code differences (diffs) between the initial automated/assisted output and the final, functional state, which will be examined in Section 4.

3.1.3 Target Services for Migration

We are focusing on two distinct services to evaluate the migration process, characterized by the following key features relevant to the migration.

Open-Source Example Service

This publicly available example service[1] comprises approximately 1 293 lines of Java code across 32 classes. Key features include REST endpoints implemented with JAX-RS (Jersey), data persistence using Hibernate ORM, database schema initialization via Liquibase, use of Dropwizard's built-in authentication/authorization mechanisms, implementation of the Data Access Object (DAO) pattern, and a comprehensive testing setup (unit, integration) using Dropwizard's test rules. The main anticipated migration challenges involve rewriting configurations, resource endpoints, adapting the data layer, and updating the extensive test suites.

Validation Strategy: For this service, migration success is validated primarily by the successful execution of its adapted preexisting test suite (approx. 20 tests covering controller, security, and DAO functionality). All adapted tests must pass. Successful database initialization via Liquibase is confirmed by the application starting correctly, and migrated health check functionality will be verified manually.

Closed-Source Proprietary Service

This internal service consists of roughly 3,500 lines of Java code spread across 27 classes. Its relevant characteristics include complex, domain-specific business logic services and custom integrations with caching mechanisms. Furthermore, it utilizes specialized logging infrastructure via private libraries and integrates metrics reporting with tools like Datadog and Rollbar, also using private libraries. The service also features asynchronous API implementations and employs custom exception handling and health check patterns. Migration complexity here is expected to be high due to intricate business logic, domain-specific constraints, and proprietary dependencies.

Validation Strategy: Migration validation relies on its preexisting suite of integration tests, which must pass after adaptation. These tests cover core APIs, business logic, and caching integrations. Additionally, the correct functioning of metrics reporting integrations

[1]Example service, Accessed: 2025-04-13, url: `https://github.com/Fossur/dropwizard-13-example-project`

(e.g., Datadog, Rollbar) will be validated manually.

3.1.4 Strategy, Principles and Validation

Our migration strategy involves three distinct phases to allow for comparison:

1. A *manual migration* to establish baseline performance and effort metrics.
2. An *LLM-assisted migration* using o3-mini with high reasoning.
3. An *AST-based migration* using OpenRewrite.

This process adheres to two core principles. The primary principle is *No Functionality Regression*, meaning the migrated service must maintain functional parity with the original Dropwizard implementation. The second principle is the *Path of Least Resistance*, prioritizing the quickest route to an operational Spring Boot service and deferring major optimizations.

To ensure adherence to the *No Functionality Regression* principle, validation is key. Finalization of a migration attempt hinges on achieving functional parity, which requires the migrated application to **compile and start cleanly**, exhibit **correct core functionality**, and crucially, **pass its existing automated test suite**. Specific validation criteria for each target service are provided in their respective descriptions (Section 3.1.3).

By analyzing the migration outcomes—validated according to these criteria—for both services across the different approaches, we aim to identify the relative advantages and disadvantages of each tooling method.

3.2 Manual Migration

This section establishes the baseline for our migration analysis by detailing the manual process of migrating two distinct Dropwizard 1.3 services to Spring Boot 2.7.x. The objective was to achieve functional parity while identifying key challenges and effort sinks inherent in the manual approach.

3.2.1 Open-Source Service

The migration of the open-source example service (32 classes, ~1 293 LOC), which used standard Dropwizard features like Hibernate, Liquibase, and JAX-RS, involved addressing several key areas.

Core Refactoring: A primary task involved replacing Dropwizard's `Application` class and its manual environment registrations within the `run` method. This was necessary due to Spring Boot's reliance on *annotation-driven component scanning* and configuration. We addressed this by eliminating the Dropwizard `Application` class and creating a new main entry point annotated with `@SpringBootApplication`. Components previously registered manually were converted to Spring Beans using annotations like `@Component` or defined in `@Configuration` classes. The main `HelloWorldApplication.java` file required a significant rewrite (93 lines modified).

Configuration: The transition from Dropwizard's *YAML/`Configuration` subclass model* to Spring Boot's `application.properties` and `@ConfigurationProperties` annotations required careful manual mapping. We created a new `@ConfigurationProperties` annotated class, removed Dropwizard-specific fields like `DataSourceFactory`, and manually added corresponding entries to `application.properties` to bind the configuration values.

Data Access: We adapted DAOs from using Dropwizard's `AbstractDAO` and `SessionFactory` to utilize Spring's `EntityManager`, a core component of *Spring Data JPA*. This involved replacing the `@UnitOfWork` annotation with Spring's `@Transactional` for transaction management. These changes necessitated significant modifications, particularly to the base `AbstractDAO.java` implementation.

Security: The security model migration involved mapping *JAX-RS security annotations* (`@RolesAllowed`, `@Auth`) to their *Spring Security equivalents* (`@PreAuthorize`, `@AuthenticationPrincipal`). The overall security configuration also had to be reimplemented from scratch to provide the same functionality.

Testing: Overhauling the test framework was a complex step. Dropwizard's *JUnit 4 Rules* (`DAOTestRule`, `ResourceTestRule`) were replaced with *Spring Boot's test*

annotations (`@DataJpaTest`, `@SpringBootTest`, `@AutoConfigureMockMvc`). Helper methods like Dropwizard's `inTransaction` lambdas were unwrapped and removed, with `@Transactional` applied at the class level where appropriate. HTTP resource tests using rule chains were manually rewritten using Spring's `MockMvc` and `RestTemplate`. Finally, tests were migrated from JUnit 4 to JUnit 5 syntax and assertions.

Outcome: The manual migration of the open-source service affected 39 files, resulting in 876 line insertions and 751 deletions. As illustrated in Figure 2, the distribution of these changes showed the largest impact concentrated on *test resources* (33.4%) and the *core application/configuration* components (18.0%), reflecting the significant effort required in adapting tests and the application's core structure.

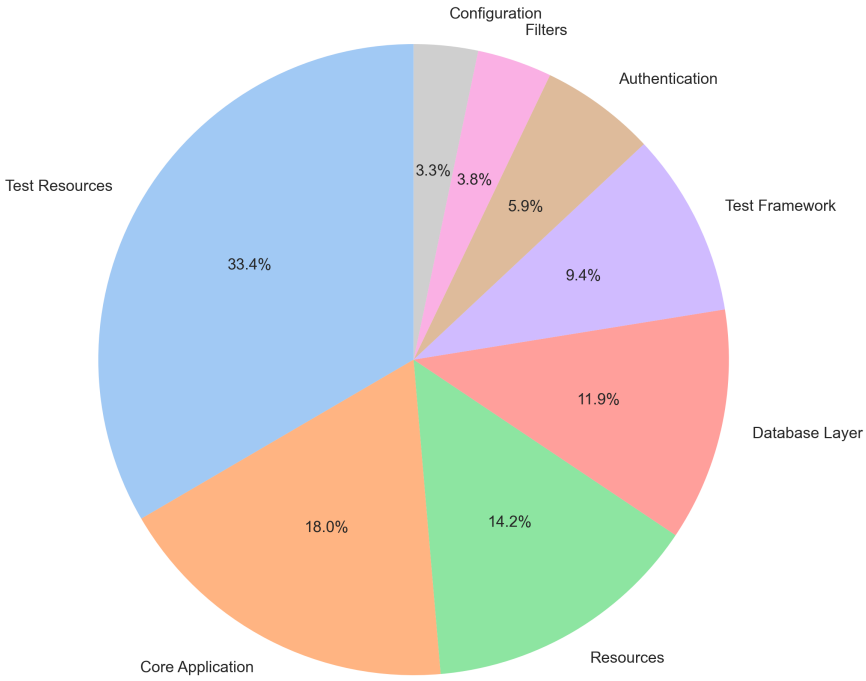

Figure 2. Distribution of changes for open-source service migration

3.2.2 Closed-Source Service

Migrating the proprietary service (27 classes, ~3,500 LOC) presented distinct challenges due to its complex business logic and custom integrations.

Proprietary Integrations: A key challenge was the careful migration of *custom interfaces and implementations* for caching, logging, and metrics (using private libraries for Datadog, Rollbar) to ensure compatibility with Spring Boot. This involved integrating seven

distinct metric beans and configuring Micrometer appropriately without disrupting existing functionality or internal contracts.

Configuration: In addition to the standard Dropwizard YAML-to-properties conversion, *proprietary configuration interfaces* required careful migration. Lombok's `@Data` annotations were introduced to configuration classes to simplify property binding in the Spring Boot context.

Complex Logic Handling: Managing *complex business logic* concentrated within a single large file (~900 lines) was particularly challenging. This demanded careful, step-by-step refactoring and extensive validation to ensure functional equivalence, as automated tools might struggle with nuanced domain logic.

API Migration: APIs and resource controllers featuring *custom exception handling* were adapted to Spring MVC (e.g., using `@RestController`). The testing framework required a similar overhaul to the open-source service, migrating from Dropwizard rules to Spring Boot test annotations and utilities.

Outcome: This migration affected 19 files, with 317 insertions and 354 deletions. The distribution of changes, shown in Figure 3, was concentrated in the entry point/configuration (29.0%) and resources/controllers (20.2%), along with significant effort in dependency and properties management (combined 28.6%). The relatively lower file count compared to the open-source service, despite having more lines of code overall, highlights the *targeted nature of changes* required around specific proprietary components and complex logic areas.

3.2.3 Overall Migration Analysis and Transition

The manual migration of both services provided a crucial performance and effort baseline, with detailed comparisons presented in Chapter 4. Several cross-cutting observations emerged from this process.

First, **common challenges** were encountered across both projects, particularly involving the intricate mapping of configuration paradigms and the correct management of dependencies between the distinct Dropwizard and Spring Boot ecosystems. A deep understanding of *framework-specific nuances* proved essential throughout.

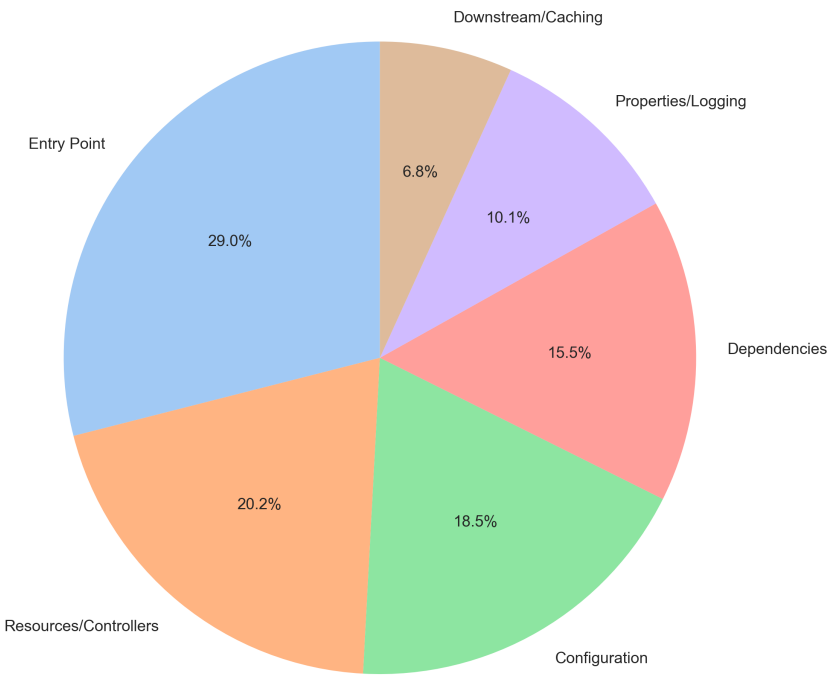

Figure 3. Distribution of changes for closed-source service migration

Second, the **migration scale** differed significantly, as shown in Figure 4. The open-source service required broader changes across 42 files (1 047 insertions, 751 deletions), whereas the closed-source migration involved more focused modifications within 18 files (341 insertions, 395 deletions). This difference indicates that Lines of Code (LOC) is not the sole driver of complexity; rather, *code complexity and proprietary integrations* significantly influence the scope and nature of the required changes.

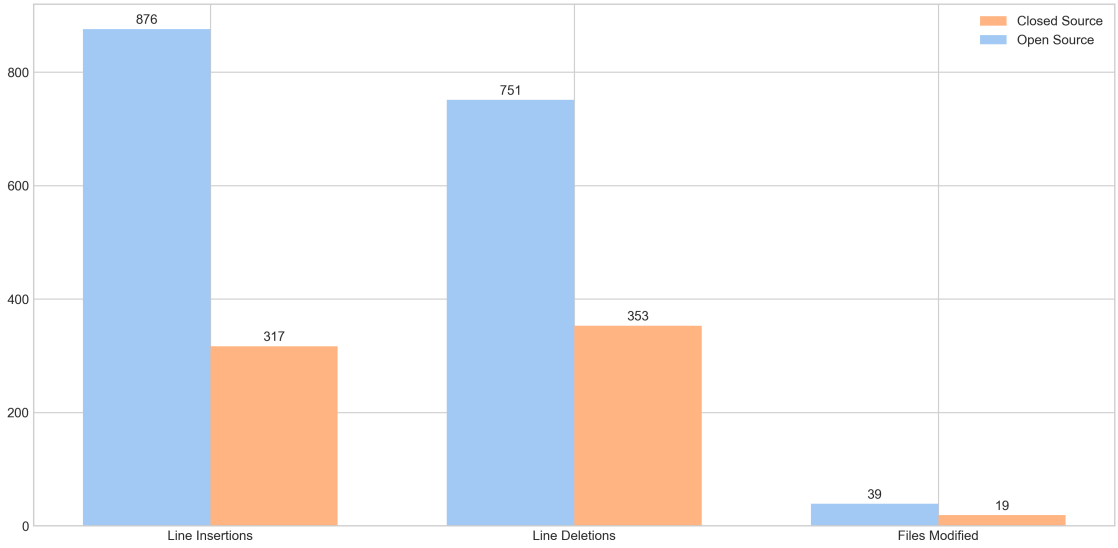

Figure 4. Comparison of migration scale between open and closed-source projects

Third, regarding **effort allocation** (depicted in Figure 5), a substantial portion of time was dedicated to *research and investigation* (55%). This involved understanding framework differences and resolving non-obvious integration issues, outweighing the time spent on purely mechanical code changes (30% syntax modifications) and subsequent debugging (15%).

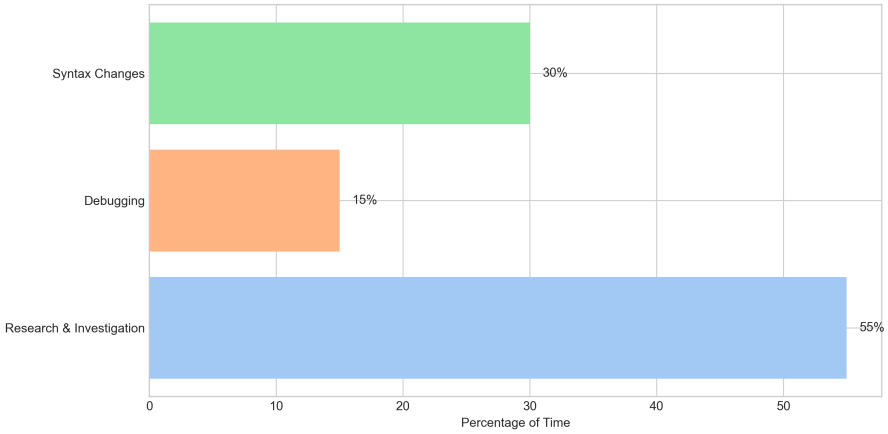

Figure 5. Distribution of time spent during migration process

Fourth, **verification** of functional parity was achieved through iterative compilation, testing, debugging, and manual checks. The specific validation approach varied; the open-source service relied heavily on adapting its comprehensive *existing Unit and Integration tests*, while the closed-source service used its more limited unit tests complemented by *external integration test suites* simulating real-world usage. Both approaches confirmed the viability of manual migration but underscored its labor-intensive nature.

Finally, these observations collectively highlight the **need for automation**. The considerable time investment, particularly in research and resolving framework differences, alongside the requirement for deep framework-specific expertise, strongly motivated the exploration of the automated methods detailed in subsequent sections. The challenges identified during manual migration directly informed the development goals and evaluation criteria for the LLM and AST-based techniques.

3.2.4 Future Considerations

While our manual migration provided valuable insights, the process highlighted the potential for leveraging automated tools such as OpenRewrite for future transitions. The complexity and time investment required for manual migration suggest that automation

could significantly streamline future framework migrations.

Based on these findings, the subsequent sections delve into two distinct automated migration strategies aimed at mitigating the challenges and reducing the effort observed in the manual process: LLM-assisted migration (Section 3.3) and AST-based migration using OpenRewrite (Section 3.4).

3.3 LLM-assisted Migration

In addition to manual refactoring and AST-based transformations, we explored whether a large language model could reduce developer effort when migrating Dropwizard applications to Spring Boot. The goal was to determine if an LLM could perform a *minimally invasive* refactor—targeting only Dropwizard-specific logic while preserving existing architecture and domain code. This scenario is especially relevant for organizations lacking deep Spring Boot expertise or wishing to accelerate migrations via high-level prompts rather than detailed manual rewrites.

For the LLM-assisted migration tasks, we utilized OpenAI's *o3-mini* model (configured for high reasoning effort), based on the selection rationale detailed in Section 2.2.5. We performed each migration in a single interactive session, allowing the model to retain context and build upon previous conversions.

The zero-shot prompt can be seen in Figure 6:

```
Hi, I want to migrate this Dropwizard class to Spring Boot.
Please remove all Dropwizard-specific logic and imports,
converting it to a Spring Boot equivalent.
Keep everything else as unchanged as possible.
If you need dependencies or properties, let me know.

\\ Code here
```

Figure 6. Example zero-shot starting prompt

When the generated output introduced compilation errors or functional regressions, we

applied focused follow-up prompts, such as `These imports do not exist` or `Can we do it in any other way?` until we got to a satisfactory result or decided that we needed to handle it manually.

3.3.1 Open-Source Service

We first tested the LLM on the open-source Dropwizard application (detailed in Section 3.1.3).

Initial Quality of Results

The initial results showed promise but also immediate challenges. The LLM effectively stripped Dropwizard imports and inserted standard Spring Boot features, such as adding `spring-boot-starter-actuator` for health checks. However, a recurring issue was its tendency to habitually replace existing DAO classes with Spring Data's `JpaRepository`, despite explicit instructions in the prompt to preserve the existing DAO pattern.

Iterative Prompts

Further interaction was necessary to refine the output. We had to specifically guide the LLM through follow-up prompts to maintain the existing `@PersistenceContext` DAOs rather than converting them to `JpaRepository`. Additionally, the initial security migration was incomplete; while the model generated an `ExampleAuthorizer` class, it omitted the core Spring Security configuration. A subsequent targeted prompt was required to elicit the necessary code implementation.

Manual Corrections

Despite iterative prompting, several manual corrections were still required. Key Spring annotations, including `@Configuration` and `@Component`, were frequently absent, necessitating repeated prompts or manual addition to resolve compilation issues. The LLM also occasionally introduced duplicate bean definitions (e.g., providing both an `@Bean` method and a `@Component` annotation for the same class), which we manually consolidated. Test configuration proved problematic initially; the model failed to enable the correct Spring Boot test contexts (`@SpringBootTest`, `@DataJpaTest`, etc.) until we provided specific stack traces as input. Finally, some references to Dropwizard's YAML configuration lingered in the generated `application.properties` file, requiring additional instructions or manual edits to remove or update them.

Ultimately, the open-source migration using this LLM-assisted approach took roughly 10 hours. While the LLM demonstrated efficiency in handling boilerplate conversion, consistent human oversight and an average of 2–5 follow-up prompts per file were necessary to address overlooked annotations, incomplete feature migrations like security, and test configuration oversights.

3.3.2 Closed-Source Service

We then tested the same LLM approach on the proprietary Dropwizard application (detailed in Section 3.1.3).

Initial Observations

Initially, the LLM correctly replaced known Dropwizard references like `io.dropwizard.Configuration` with partial Spring Boot configurations, including `@SpringBootApplication` and `@ConfigurationProperties`. However, it also attempted to replace the service's proprietary metrics implementations with standard Micrometer or Actuator beans, requiring multiple corrective prompts to reinstate the original library usage and configuration.

Over-Invasive Changes

A significant challenge was the LLM's tendency towards over-invasive refactoring, despite prompts requesting minimal modifications. It frequently renamed classes without instruction, reorganized method structures, and, in some files, removed nearly all existing comments, losing potentially valuable context. Defaulting perhaps to perceived *best practices*, the model sometimes generated code that was more idiomatic to Spring Boot but diverged significantly from the original application's logic or established domain approach.

Corrections and Workarounds

Addressing the LLM's output for the closed-source service required numerous corrections. The model repeatedly removed or replaced the existing metrics framework configuration, forcing us to manually revert these changes or provide highly specific prompts. Several references to Dropwizard classes persisted in the code, necessitating iterative prompting for their complete removal.

A significant issue arose with migrating JAX-RS resource definitions. Two primary paths exist: 1) utilizing Spring Boot's Jersey integration (`spring-boot-starter-jersey`)

to reuse existing JAX-RS annotations with minimal changes, or 2) performing a full migration to Spring MVC annotations (`@RestController`, `@GetMapping`, etc.). The LLM defaulted to the more invasive Spring MVC migration path. While this approach is valid, it failed dramatically on a large, complex resource file (~900 lines), generating unusable code with missing method parameters, comments and general necessary bits of business logic. Even when prompted for alternatives (e.g., `Is there any other optimal way?`), the model did not suggest the simpler Jersey compatibility approach until explicitly directed towards `spring-boot-starter-jersey`. Consequently, we ultimately reverted the LLM's changes to the large resource file, opting for the Jersey compatibility setup which required significantly less modification, highlighting the LLM's difficulty in identifying the *path of least resistance* for complex components.

Furthermore, default Dropwizard exception mappers were replaced by Spring's `@ExceptionHandler` mechanism which, while functionally valid, conflicted with existing business-specific error handling flows, necessitating a reversion to a custom scheme. Lastly, the LLM discarded comments present in the original code, which had to be reinstated manually.

3.3.3 Key Findings and Recommendations

Our experiences with LLM-assisted migration across both services highlighted several key strengths, limitations, and practical considerations.

Strengths of LLM Migration

The primary strengths observed were the LLM's ability for *rapid Dropwizard removal*, quickly eliminating references to framework-specific classes and attempting rewrites in a Spring Boot context. It also proved efficient at *boilerplate generation*, producing initial versions of entry-point files, configurations, and potential `pom.xml` modifications. Furthermore, the LLM demonstrated good *prompt responsiveness*; when presented with specific compile-time errors or short clarification requests, it could typically correct its mistakes in subsequent attempts.

Recurring Limitations

Several limitations recurred throughout the process. *Hallucinations and over-refactoring* were common issues, where the LLM might rename classes, rearrange code unnecessarily,

or strip comments without explicit instruction. Migrations involving *partial security and DAO implementations* were frequent, often requiring additional prompts or manual intervention to preserve specialized patterns like `@PersistenceContext` over the default suggestion of `JpaRepository`. Handling *proprietary libraries* proved challenging, as domain-specific or private code often confused the LLM, yielding incomplete or incorrect solutions. *Dependency handling, in general, was unreliable;* while the LLM could add obvious Spring dependencies, it often missed necessary project-specific or transitive dependencies, and incorrectly handled proprietary ones, meaning dependency management still required thorough manual review and completion. Finally, the quality of migration degraded significantly for *large files* (approx. 500+ lines), which suffered more frequently from missing logic, invalid constructors, incorrectly altered method parameters, removed comments, and incomplete import statements. *Our observations suggest this is likely due to context window limitations or processing difficulties with larger inputs, as providing the LLM with smaller, more focused code chunks generally yielded better, more complete results.*

Practical Recommendations

Based on these findings, we recommend several practical strategies for effectively using LLMs in similar migration tasks. *Process code in smaller, manageable chunks rather than feeding entire large files to the model*, especially for complex classes, to improve the quality and completeness of the output. Employ an *iterative prompt strategy*, providing an initial high-level prompt for broad changes, then using short, targeted follow-up prompts to address specific errors or refine outputs (e.g., "Only fix the test configuration"). When dealing with proprietary code, provide *example classes* demonstrating typical usage if possible, or explicitly instruct the model not to remove or replace specific dependencies. Maintain *version consistency* by double-checking the LLM's recommended library versions, as these can sometimes vary between prompts. Crucially, perform *final verification* by always compiling, running tests, and conducting code reviews after LLM generation; providing compile errors back to the LLM can sometimes allow it to self-correct. Lastly, anticipate manually adjusting *logging and properties*, especially environment-specific or proprietary logging configurations, as these are often handled inaccurately by current models.

3.3.4 Overall Assessment

Both the open-source and closed-source migrations demonstrably benefited from LLM assistance in terms of migration speed, primarily through the reduction in effort required for repetitive or boilerplate transformations. The open-source service migration was completed in approximately 10 hours, which we estimate is about half the time a purely manual approach might have taken without prior Spring Boot expertise. The closed-source service migration, finished in 7.5 hours, was similarly accelerated despite the challenges posed by extensive proprietary logic, including a large resource file that ultimately required manual restoration.

In summary, a large language model can substantially speed up certain stages of a framework migration by handling routine conversions and reacting intelligently to iterative corrections. Nevertheless, our experience underscores that it must be closely guided and its output rigorously verified to avoid over-modification, omitted parameters and comments, or erroneous replacements of existing logic. Organizations evaluating LLM-based migrations should carefully balance the model's ability to rapidly remove old framework code against the critical need to preserve proprietary libraries, specific domain logic, and carefully tuned architectural structures through diligent oversight and targeted prompting.

3.4 AST-based Migration

In this chapter, we describe how we used **OpenRewrite** to automate a Dropwizard 1.3–to–Spring Boot migration. We wanted to determine how much of it could be replicated by purely **AST-based refactoring** tools and what are the most effective ways to use this tool. Our core question was how *reasonably* we could automate the transition, striking a balance between the following factors:

Specifically, we needed to consider *cost-efficiency*, as we did not want to spend excessive time writing custom transformation logic for potentially unique cases, while also aiming for *reusability* by generalizing the recipes for use across other projects beyond just a single service. Furthermore, we acknowledged the *coverage* limitations, recognizing that while some framework aspects map neatly, others (like Security implementations) might still require extensive manual intervention. Ultimately, we set out to create a **proof-of-concept**

that demonstrates how developers could script a large portion of Dropwizard's features into an idiomatic Spring Boot application.

3.4.1 Recipe Structure

In order to systematically transform our Dropwizard application into a Spring Boot–based service, we composed a series of modular OpenRewrite recipes. At the highest level, the migration logic is centered around a primary composite recipe, `ee.taltech-.CompositeDropwizardToSpringBoot`. This recipe declares a set of `recipeList` entries referencing the subordinate modules, each targeting a specific functional area of the Dropwizard to Spring Boot transition, as depicted in Figure 7.

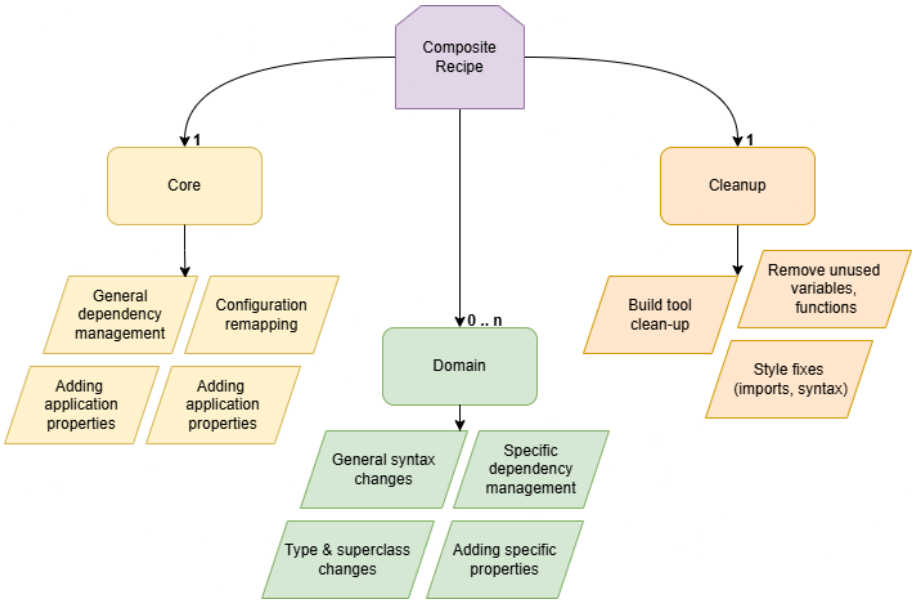

Figure 7. Composite recipe structure: featuring core, domain and cleanup recipes

The primary modules included in the composite recipe's `recipeList`, orchestrating the migration from configuration and core infrastructure to specific features like security and testing, were:

1. `Core Setup:` Handles the fundamental application structure, including entry point setup and main dependencies.
2. `Configuration Migration` Converting the main Dropwizard configuration file to Spring format.
3. `Monitoring and Health Checks:` Adapting Health checks and actuators to Spring Boot.

4. `Security Framework`: Migrating various annotations, adding a scaffolding file that is ready to be implemented.
5. `Hibernate`: Updates the existing DAO interface to be able to handle Spring's Hibernate injection mechanisms.
6. `Resource`: Adapts REST endpoints and resources to Spring's Jersey-starter, which allows to maintain compatibility with existing JAX-RS annotations.
7. `Tasks and Commands`: Converts background processes and administrative commands to equivalent Spring Boot mechanisms.
8. `Tests`: Updates the testing infrastructure to use Spring Boot's testing mechanism like RestTemplate.
9. `Metrics`: (Closed-source service only) Updates types and dependencies for metrics.
10. `General Cleanup`: Handles general code cleanup like removing leftover variables or methods, changing miscellaneous classes.

This layered organization, built upon the composite recipe structure, makes the migration procedure highly customizable. Developers can run the entire composite script for a full migration attempt or invoke specific child recipe modules selectively if they prefer to validate each area of refactoring step by step. The structure also inherently lends itself to easy extension; introducing additional transformations—for example, specialized security refinements or advanced resource conversions—can be done by adding new modules without modifying the rest of the established pipeline.

A key challenge was migrating configuration values from Dropwizard's flexible YAML structure to Spring Boot's property-based system. We determined that automatically parsing the existing Dropwizard YAML files exhaustively to map every possible key and value permutation reliably within a general-purpose OpenRewrite recipe was impractical due to the potential complexity and variability of YAML structures.

Therefore, we adopted a strategy focused on scaffolding and manual value transcription. Instead of parsing the old YAML, we opted to include *starter* configuration recipes (e.g., `AddHibernateConfiguration`, `AddSecurityConfiguration`) designed to append essential Spring Boot properties to the target `application.properties` file. These recipes add critical property keys using placeholder or sensible default *dummy values*, ensuring the migrated application has the necessary configuration structure. This allows developers

to subsequently fill in their environment-specific values manually, often referencing their old YAML file, without needing to recreate the property keys from scratch. An example snippet from a recipe adding Hibernate-related properties is shown in Figure 8.

```
recipeList:
- org.openrewrite.properties.AddProperty:
    property: spring.datasource.url
    value: jdbc:h2:mem:mydb # Default/placeholder value
    delimiter: "="
- org.openrewrite.properties.AddProperty:
    property: spring.datasource.driverClassName
    value: org.h2.Driver # Default/placeholder value
    delimiter: "="
- org.openrewrite.properties.AddProperty:
    property: spring.jpa.hibernate.ddl-auto
    value: validate # Default/placeholder value
    delimiter: "="
```

Figure 8. Example OpenRewrite recipe actions adding placeholder properties for Spring Data JPA

These helper property additions were bundled logically within the relevant recipe modules; for instance, the `MigrateHibernate` module included the actions shown above to add standard Spring Data JPA properties, potentially with comments guiding the developer on where to find original values.

Separately, for instances where migrating the syntax of certain Java classes or features proved too complex or extensive to transform reliably via AST manipulation alone, **we created template source files** using the `org.openrewrite.text.CreateTextFile` recipe instead. Figure 9 illustrates this approach, showing how a minimal skeleton `SecurityConfig.java` class was generated. This class provides the necessary Spring annotations (`@Configuration`, `@EnableWebSecurity`, etc.), but project-specific authentication and authorization logic must be implemented manually.

The parametrized values are filled out using built-in Gradle property substitution before compiling the recipe. Although the file includes key Spring annotations (e.g.,

```
- org.openrewrite.text.CreateTextFile:
    relativeFileName: ↩
  "${sourcePath}/${classPath}/configuration/SecurityConfig.java"
    fileContents: |
        package ${mainPackage}.configuration;

        # imports omitted for clarity

        @Configuration
        @EnableWebSecurity
        @EnableMethodSecurity
        public class SecurityConfig {

        }
```

Figure 9. Adding security configuration scaffolding via recipe

`@EnableWebSecurity`), it remains a template that must be adjusted for project-specific logic. If multiple services share the same security requirements, one could instead package this class in a dedicated library.

In some cases, where to add newly created files (e.g., `SecurityConfig.java`) was not obvious. We as humans intuitively place controllers, services, or configurations in logical package structures, but scripts instead can only rely on placeholders or user-supplied configuration paths. We resolved this by making the package name configurable, acknowledging that the final code arrangement might require manual refinement.

3.4.2 Dependency Management and Exclusions

Reducing Dropwizard dependencies within the project was another priority. OpenRewrite provides for dependency management, which makes it easy to codify addition (Figure 10) and removal (Figure 11).

```
- org.openrewrite.maven.AddDependency:
    groupId: org.springframework.boot
    artifactId: spring-boot-starter-actuator
    version: "${springBootVersion}"
    onlyIfAbsent: true
```

Figure 10. Adding a new Spring Boot dependency

```
- org.openrewrite.maven.ExcludeDependency:
    groupId: io.dropwizard
    artifactId: dropwizard-jersey
```

Figure 11. Excluding an unused Dropwizard dependency

Such changes might seem straightforward, but they encapsulate the decisions about which Dropwizard packages require removal and which Spring Boot counterparts should replace them, effectively saving the developer valuable research time. By parameterizing the Spring Boot version (e.g., `${springBootVersion}`), we ensure that developers can override this value externally if they prefer a different Spring Boot release.

Migrating Dropwizard applications to Spring Boot often involves handling JAX-RS resources, typically defined using annotations like `@Path`, `@GET`, and `@Produces`. To avoid an immediate requirement to rewrite these endpoints using Spring MVC annotations, we incorporated the *spring-boot-starter-jersey* dependency. This starter allows existing JAX-RS resources to function within the Spring Boot environment.

Spring Boot's JAX-RS auto-configuration aids this integration by detecting classes annotated with `@Path`. For these resources to be managed by Spring's IoC container, they also need to be recognized as Spring components.

One method to bridge Jersey and Spring is through explicit configuration. A configuration class extending Jersey's ResourceConfig, such as the example in Figure 12, can programmatically register Spring beans annotated with `@Path`. This class injects the Spring

ApplicationContext, retrieves all beans carrying the `@Path` annotation, and registers them with Jersey.

Jersey Configuration Example

```
@Configuration
public class JerseyConfig extends ResourceConfig {

    public JerseyConfig(ApplicationContext applicationContext) {
        Map<String, Object> beans =
            applicationContext.getBeansWithAnnotation(Path.class);
        beans.values().forEach(this::register);
    }
}
```

Figure 12. Jersey-annotated Controllers initialization file

There are two primary ways to incorporate this `JerseyConfig` class into target projects. Our implementation embedded it within a shared helper library, allowing easy reuse across multiple microservices. Alternatively, an OpenRewrite recipe could be employed to add the `JerseyConfig.java` source file directly into each project's codebase. However, using AddFile might necessitate manual adjustments to the file's target path, especially within complex multi-module repositories, to ensure it resides in the appropriate module and package.

To automate and further streamline the migration, we also developed a recipe that adds the Spring `@Component` annotation to any existing class already annotated with JAX-RS's `@Path` - which would allow the configuration in Figure 12 to pick it up. By marking these JAX-RS resources as Beans, they become automatically discoverable via Spring Boot's component scanning mechanism.

These configuration-based and automated annotation approaches ensure that JAX-RS resources are seamlessly integrated into the Spring container. While resources could still be registered manually, adopting these techniques significantly reduces the migration effort compared to the traditional Dropwizard pattern of individual registrations (e.g., `environment.jersey().register(...)`).

Future Migration to Spring MVC

Eventually, teams might wish to replace JAX-RS annotations with native Spring MVC constructs like `@RestController` and `@GetMapping`. Because these changes often demand rewriting method signatures, path definitions, and response-handling logic, we decided that a separate recipe (or a purely manual pass) would be more realistic than trying to handle all possible JAX-RS constructs in a single automated script. Nonetheless, the flexible design of our composite script means that such a specialized recipe could be appended if necessary.

3.4.3 Configuration

Although Dropwizard and Spring Boot each rely on externalized configuration, their methods differ considerably. Where Dropwizard reads a YAML file into a subclass of `Configuration`, Spring Boot typically uses a combination of `application.properties` (or `application.yml`) and annotation-driven property binding. We approached this reorganization in two parts: refactoring the *main entry point* file, and migrating the *main configuration* class.

Dropwizard set-up

In a typical Dropwizard project, the `Application<T>` class provides a `run(...)` method that registers bundles, commands, tasks, or resources (Figure 13).

```
1  public class HelloWorldApplication extends
       Application<HelloWorldConfiguration> {
2      @Override
3      public void run(HelloWorldConfiguration configuration,
4                      Environment environment) {
5          environment.healthChecks().register("template",
6              new TemplateHealthCheck(configuration.getTemplate()));
7          environment.jersey().register(new PeopleResource(...));
8          environment.admin().addTask(new EchoTask());
9      }
10 }
```

Figure 13. Example Dropwizard resource initialization

3.4.4 Rewiring the Main Entry Point and Configuration Classes

Migrating the application's core structure requires bridging the gap between Dropwizard's inheritance-based model (`io.dropwizard.Application`) and Spring Boot's composition model (`@SpringBootApplication`, component scanning, auto-configuration). Our OpenRewrite migration transforms the main Dropwizard application class itself into the primary Spring Boot entry point, modifying the existing class directly through several automated steps:

First, the extends `io.dropwizard.Application` clause and related method overrides (like initialize) are removed from the existing Dropwizard application class (e.g., `HelloWorldApplication`). Notably, the original run method signature, while no longer an override, is temporarily preserved for analysis in a subsequent step. This initial change decouples the class from Dropwizard framework's lifecycle specifics.

Second, the `@SpringBootApplication` annotation is added directly to the modified class (`HelloWorldApplication`). This single annotation conveniently enables core Spring Boot features by implicitly including `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan`, marking the class as the primary configuration and component-scanning hub.

With the class now acting as a Spring @Configuration source, the logic previously contained within the original Dropwizard run method body is processed. Our recipes parse this body to identify Dropwizard-specific component registrations (e.g., `environment.admin().addTask(...)`, `environment.healthChecks().register(...)`, `environment.jersey().register(...)`). These registrations are then automatically translated into corresponding `@Bean` methods defined directly within this same `@SpringBootApplication` annotated class. This step effectively converts the application's component wiring from Dropwizard's imperative style to Spring's declarative bean definitions (seen in Figure 14).

The final step for completing the structure of this entry point class is adding the standard `public static void main(String[] args)` method. This method invokes `SpringApplication.run(...)`, providing the necessary mechanism to bootstrap the now Spring-native application - the class name does need to be manually adjusted here

```
// Dropwizard explicit resource definition
MyTask myTask = new MyTask(configuration.getSomeSetting());
environment.admin().addTask(myTask);

// Spring Boot Bean equivalent
@Bean
public MyTask myTaskBean(HelloWorldConfiguration appConfig) {
    // Instantiation logic is preserved in the Bean definition
    return new MyTask(appConfig.getSomeSetting());
}
```

Figure 14. Dropwizard resource conversion to Spring Bean

after.

As a concluding cleanup phase for this class transformation, dedicated cleanup recipes are executed (in the General Cleanup module). These recipes specifically target and remove remnants of Dropwizard framework by identifying code elements based on Dropwizard package names (e.g., `io.dropwizard.*`). This ensures that any now-unused variables and methods related to Dropwizard classes—particularly those involved in the old initialization process—are cleanly removed, leaving a more idiomatic Spring Boot entry point class.

This parallel transformation approach—adding stereotype annotations (`@Component`) to discovered classes while also translating explicit registrations from the run method into `@Bean` methods can lead to bean definition conflicts detected by Spring. Simply removing the explicit `@Bean` declarations automatically is risky, as it might omit necessary wiring or configuration from the original run method. We tried to resolve bean-definition conflicts automatically by first detecting any existing `@Component` (or equivalent) annotations; the strategy worked in some cases, but proved only partly reliable because the underlying type system did not always expose the necessary metadata.

Migrating the Dropwizard Configuration Class

Dropwizard configuration typically uses a class extending `io.dropwizard.Configuration` with fields mapped to YAML entries (seen in Figure 15).

```java
public class HelloWorldConfiguration extends Configuration {
    @NotEmpty
    private String template;

    @Valid
    @NotNull
    private DataSourceFactory database; // Dropwizard-specific type

    // ...other configuration fields with getters and setters
}
```

Figure 15. Partial Dropwizard Configuration file example

Our recipes refactor this class for Spring Boot property binding through a sequence of automated steps. Initially, a custom recipe removes the `extends io.dropwizard.Configuration` clause and any now-invalid `@Override` annotations. Subsequently, the class is prepared for Spring Boot property binding by adding `@ConfigurationProperties` and Lombok's `@Data` for the main class and all inner classes - this will generate set and get methods, which will allow Spring Boot to assign the properties. The `@ConfigurationProperties` annotation enables type-safe binding from external sources like application.properties, while `@Data` auto-generates the getters and setters required by Spring's binding mechanism. Using a distinct prefix helps namespace application-specific settings.

Finally, fields, getters, and setters tied to Dropwizard-specific types (like DataSourceFactory or ViewRendererConfiguration) are removed in the end with package matching and equivalent properties are assigned in the application.properties. The result is a standard Spring Boot `@ConfigurationProperties` bean.

This automated refactoring process transforms the core Dropwizard Application and Configuration classes into their Spring Boot equivalents, establishing the standard `@SpringBootApplication` entry point and utilizing `@Configuration` and `@ConfigurationProperties` for bean management and settings, respectively. While the approach streamlines the migration, potential challenges like bean definition conflicts may

necessitate manual review and adjustments to ensure correctness and optimal configuration.

3.4.5 Security

Migrating the security model from Dropwizard to Spring Security required transforming relevant annotations, as the underlying framework mechanisms differ significantly (custom Dropwizard Authenticators/Authorizers vs. Spring Security filters). Our OpenRewrite recipes focused on automating the common annotation changes involved.

Specifically, the recipes handled key security annotation migrations by mapping Dropwizard/JAX-RS annotations to their Spring Security equivalents, leveraging the strategies outlined in Section 3.4.9. Common transformations included converting `@RolesAllowed("ROLE")` expressions to `@PreAuthorize("hasRole('ROLE')")`, changing `@PermitAll` to `@PreAuthorize("permitAll()")`, and replacing Dropwizard's `@Auth` annotation on method parameters with Spring Security's `@AuthenticationPrincipal`. While these recipe steps addressed the straightforward annotation replacements, we noted during the process that the logic within the method bodies using these annotations often required subsequent manual review and updates to fully align with Spring Security's principal object and authorization context conventions. The setup of the basic Spring Security configuration class itself was handled via the file generation approach seen in Figure 9.

We deliberately kept this configuration minimal, providing only the essential annotations (`@EnableWebSecurity` and `@EnableMethodSecurity`) without prescribing specific authentication rules or user stores. This approach allows teams to implement their own authentication logic while ensuring the basic security infrastructure is in place.

3.4.6 Hibernate and Data Access Layer

A significant part of the migration involved transforming Dropwizard's Hibernate integration to Spring's Data JPA approach. This required changes to both dependencies and code structure, particularly around Dropwizard's commonly used `AbstractDAO` pattern.

Custom AbstractDAO Implementation

To decouple projects completely from the `dropwizard-hibernate` dependency and its numerous transitive dependencies, we recreated a compatible `AbstractDAO` helper class

within our migration support library. We maintained the same public method signatures and core functionality (like `persist()`, `get()`, named queries) familiar to developers using the Dropwizard pattern, but its internal implementation was rewritten using Spring's JPA `EntityManager` instead of Dropwizard's `SessionFactory`. This strategy allowed existing DAO implementations extending the original Dropwizard `AbstractDAO` to function with minimal code changes after migration, while successfully eliminating the underlying Dropwizard dependency.

The migration process leveraging this custom base class involved three main automated transformations performed by our recipes:

1. Changing the superclass of existing DAOs from `io.dropwizard.hibernate.AbstractDAO` to our custom, Spring-based implementation.
2. Adding `@Repository` annotations for Spring to be able to pick it up.
3. Adding the necessary Spring JPA `EntityManager` field, typically injected via `@PersistenceContext` or constructor injection, to the DAOs.
4. Updating the constructors of the implementing DAO classes to remove the original `SessionFactory` parameter and accept the `EntityManager` if using constructor injection.

Transaction Management

We replaced Dropwizard's method-level `@UnitOfWork` annotation, used for defining transactional boundaries, with Spring's standard `@Transactional` annotation. Our recipes performed this replacement on relevant service or DAO methods. We also added `@Transactional` at the class level to existing DAO implementation classes (often annotated with `@Repository`) to ensure consistent default transaction behaviour across all data access methods, aligning with common Spring practices.

This overall transformation, centered around the custom `AbstractDAO`, preserved the familiar DAO pattern while fully integrating data access operations with Spring's transaction management and JPA infrastructure, generally requiring minimal manual intervention for basic functionality. It is worth noting that a full migration to Spring Data JPA repositories (using interfaces extending `JpaRepository`) is another valid approach for simpler services or those with few existing complex DAO implementations. However, that alternative path would require completely rewriting existing DAO classes into repository interfaces,

potentially necessitating significant manual effort to recreate custom queries and logic currently embedded within the DAOs. Our chosen approach prioritized minimizing invasive changes to existing data access logic, aligning with the *Path of Least Resistance* principle, particularly beneficial for services with numerous or complex pre-existing DAO implementations.

3.4.7 Tests

Migrating the Dropwizard test framework to Spring Boot's testing model frequently proved to be one of the more intricate tasks during our migration efforts. Dropwizard relies heavily on JUnit 4 rules such as `DAOTestRule`, `ResourceTestRule`, and `DropwizardAppRule`, each providing specialized logic for setting up test environments like in-memory databases or partial server initialization, which needed replacement with Spring Boot equivalents.

Replacing Rules with Spring Boot Annotations

Our script analyzed class-level fields annotated with `@Rule` to detect common Dropwizard test rules. Migrating these involves mapping them to suitable Spring Boot test annotations. Spring Boot offers various annotations designed to load different slices of the application context, potentially allowing tests targeting specific layers (like the web layer or data layer) to run faster by not initializing the entire application. Common conceptual mappings include:

- `DropwizardAppRule` → `@SpringBootTest` (Full application context)
- `ResourceTestRule` → `@SpringBootTest` + `@AutoConfigureMockMvc` (Full context + MockMvc for web testing) or potentially `@WebMvcTest` (MVC slice only)
- `DAOTestRule` → `@DataJpaTest` (Data JPA slice only)
- `DropwizardClientRule` → `@WebMvcTest` (MVC slice only, for client-side testing against mocked MVC)

However, the optimal choice depends critically on the application's architecture and the overall migration strategy. Since we opted to maintain JAX-RS compatibility using `spring-boot-starter-jersey` (Section 3.4.2) rather than migrating fully to Spring MVC, annotations focused on the Spring MVC stack (`@WebMvcTest`, `@AutoConfigureMockMvc`) were generally unsuitable for testing our Jersey-based resources. While using test slices can improve test execution speed, ensuring functional

parity within the integrated Jersey environment necessitated loading the full application context.

Therefore, our primary automated strategy involved mapping the broader integration test rules (e.g., `DropwizardAppRule`, `ResourceTestRule`) directly to `@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)`. Although this approach boots the entire context and may run slower than a focused slice test, it ensures the full application environment, including the deployed Jersey resources, is available for testing via HTTP client calls (like `TestRestTemplate`), guaranteeing functional correctness. For tests focused solely on the data layer (`DAOTestRule`), mapping to the more efficient `@DataJpaTest` slice remained the appropriate strategy. We implemented this logic using a custom, extendable recipe that inspected the rule type and inserted the corresponding primary Spring annotation (`@SpringBootTest` or `@DataJpaTest`).

Adapting Test Logic

Beyond replacing the rules, adapting the test logic itself involved several automated steps. Many Dropwizard DAO tests utilize the `inTransaction` helper method provided by `DAOTestRule` to manage transaction boundaries within a test method (Figure 16).

```
\\ Before
daoTestRule.inTransaction(() -> {
    personDAO.create(new Person("Jeff", "The plumber"));
    // ... assertions or other operations within transaction ...
});

\\ After
personDAO.create(new Person("Jeff", "The plumber"));
// ... assertions or other operations within transaction ...
```

Figure 16. Example Dropwizard DAO test using `inTransaction` lambda.

Under Spring Boot testing, particularly with `@DataJpaTest`, transaction management is typically handled automatically or via the `@Transactional` annotation, rendering the `inTransaction` lambda unnecessary. To handle this, we wrote a custom Java visitor, `MethodLambdaExtractor`, specifically designed to find these

`daoTestRule.inTransaction(...)` calls, extract the code from within the lambda body, and place it directly into the containing test method, effectively removing the obsolete wrapper.

We also included recipe steps to migrate common Mockito mocking patterns. Fields initialized via `Mockito.mock(SomeClass.class)` were transformed where appropriate to use Spring Boot's `@MockBean` annotation, integrating mocking with the Spring application context used in the tests.

Migrating the HTTP request invocations within resource tests, typically expressed as chain-of-method calls on the Dropwizard test rule instance (e.g., `RULE.target("/someEndpoint").request().post(...).getStatus()`), required translation to use Spring's `TestRestTemplate` API interacting with the `@SpringBootTest`-managed application context. Due to the complexity and variability in the original invocation patterns (including different ways of setting paths, headers, entities, and asserting responses), a perfect AST-based transformation proved highly challenging. As a pragmatic solution, **we developed a converter that parsed the original Dropwizard rule invocation syntax, partially using regular expressions, to extract key information**. This converter was designed to generate equivalent `TestRestTemplate` calls, capable of mapping common request methods, adding standard headers (like authentication or JSON content types), and handling basic input/output entity types. This approach provided a *good enough* level of automation for many standard test cases. However, its reliance on specific syntax patterns meant that some original invocations were left unchanged by the automated process due to OpenRewrite's parser matching issues and required subsequent manual migration to the `TestRestTemplate` API.

Migrating the HTTP request invocations within resource tests [. . .] proved difficult within the project's scope. While our recipes could handle some simple patterns, complex test logic involving chained client calls and specific assertions often required manual rewriting to use the `TestRestTemplate` API effectively. Determining the exact beans needed for injection (like `TestRestTemplate` or `ObjectMapper` for `@SpringBootTest`, or `TestEntityManager` for `@DataJpaTest`) was generally straightforward based on the primary test annotation, but sometimes required manual additions depending on the specifics of the adapted test logic.

JUnit 4 to JUnit 5 Migration

Lastly, we leveraged existing standard OpenRewrite (`org.openrewrite.java.testing.-junit5`) recipes to convert tests from JUnit 4 constructs (e.g., `@Test(expected = ...)`, `@Before`, `@After`, `Assert.assertEquals`) to their modern JUnit 5 equivalents (e.g., `assertThrows`, `@BeforeEach`, `@AfterEach`, `Assertions.assertEquals`). This step, while not strictly part of the Dropwizard to Spring Boot framework migration itself, was included because it frequently accompanies such modernizations, and Spring Boot's testing support integrates seamlessly with JUnit 5.

We observed that this combined testing migration approach successfully automated many of the mechanical tasks, such as removing Dropwizard rules, adopting the primary Spring test annotations (`@SpringBootTest`, `@DataJpaTest`), converting common mocking patterns, inlining `inTransaction` calls, and updating JUnit syntax. However, the intricate logic within test bodies, especially HTTP resource test invocations and assertions requiring the `TestRestTemplate`, frequently remained a significant component requiring manual implementation or adjustment.

3.4.8 Class Hierarchy Transformations

We developed two key recipes for managing class hierarchies during migration: `ChangeSuperclassRecipe` and `RemoveSuperclassRecipe`. For the latter, we also made a modified version to be able to remove superclasses by package for easier cleanup.

ChangeSuperclassRecipe Parameters

1. `targetClass`: Specifies the fully qualified name of the class to transform
2. `newSuperclass`: Defines the fully qualified name of the new parent class
3. `keepTypeParameters`: When true, preserves generic type parameters during the transformation
4. `convertToInterface`: Changes inheritance from `extends` to `implements`, useful when migrating to interface-based designs
5. `addAbstractMethods`: Automatically generates stub implementations for new abstract methods, throwing `UnsupportedOperationException` to mark areas needing attention
6. `removeUnnecessaryOverrides`: Removes method overrides that become redundant after the superclass change

Migration Examples

Basic Method Scaffolding: The `addAbstractMethods` parameter proved useful for automatically generating required method stubs, as shown in Figure 17.

```
// Original abstract class
abstract class AbstractParent {
    abstract void doSomething(String input);
}

// Original child class
class Child extends AbstractParent {
}

// After transformation with addAbstractMethods=true
class Child extends AbstractParent {
    @Override
    public void doSomething(String input) {
        throw new UnsupportedOperationException();
    }
}
```

Figure 17. Example method stub generation

In summary, the development of the `ChangeSuperclassRecipe` and `RemoveSuperclassRecipe` provided essential tooling for managing structural changes related to class inheritance during the migration. The configurable parameters, particularly `addAbstractMethods`, offered a pragmatic approach to handling methods with significantly altered signatures or logic, balancing automation (stub generation) with the need for manual refinement where overly complex recipe logic would be cost-prohibitive or generally not feasible. These recipes were instrumental in adapting core Dropwizard patterns like `ConfiguredCommand` and `HealthCheck` to their Spring Boot equivalents (`CommandLineRunner` and `HealthIndicator`, respectively), and also facilitated the cleanup or complete reimplementation of classes better suited to native Spring patterns, contributing significantly to the overall framework transition.

Command Line Migration

When migrating Dropwizard commands to Spring Boot, we used the recipe in Figure 18.

```
- ee.taltech.general.ChangeSuperclassRecipe:
    targetClass: io.dropwizard.cli.ConfiguredCommand
    newSuperclass: org.springframework.boot.CommandLineRunner
    convertToInterface: true
    keepTypeParameters: false
    addAbstractMethods: true
    removeUnnecessaryOverrides: true
```

Figure 18. Recipe to change superclass with different available options

This transformation converted Dropwizard command classes into Spring Boot's `CommandLineRunner` interface, with the original command logic moving into the `run(...)` method.

Health Check Migration

For health checks, we opted for manual method migration rather with various methods - this worked well because all the inner method invocations and return types mapped well to Spring Boot equivalents, and thus there was no need to generate existing method stubs (Figure 19).

```
// Dropwizard version
class CustomHealthCheck extends HealthCheck {
    @Override
    protected Result check() {
        return Result.healthy();
    }
}

// Spring Boot version (manual migration)
class CustomHealthCheck implements HealthIndicator {
    @Override
    public Health health() {
        return Health.up().build();
    }
}
```

Figure 19. Health check implementation

Class Removal with RemoveSuperclassRecipe

In some cases, rather than transforming classes directly, it made more sense to remove their Dropwizard inheritance entirely and subsequently reimplement their functionality using a more idiomatic Spring pattern. For instance, Dropwizard's `PostBodyTask` functionality, often used for administrative actions, was generally better suited to implementation as a standard Spring `@RestController` endpoint. The `RemoveSuperclassRecipe` facilitated the cleanup of such classes targeted for reimplementation.

1. Removes the target class's superclass relationship
2. Cleans up associated override methods
3. Allows developers to reimplement functionality using framework-appropriate patterns

This approach strategically avoided creating complex, project-specific transformations that might not generalize well across different codebases.

3.4.9 Annotation Migration Strategy

Converting annotations represented a central challenge in our Dropwizard to Spring Boot migration. While Dropwizard relies on JAX-RS standards and annotations like `@RolesAllowed` and `@Auth`Spring Boot uses its own ecosystem including `@RestController`, `@PreAuthorize`, and `@AuthenticationPrincipal`. We established three key principles for this migration: minimize repetitive work, preserve existing project conventions, and focus on widely-used transformation patterns.

Core Migration Patterns

Our annotation migration strategy employed three main techniques:

1. **Inheritance-Based Detection** We identified migration candidates through their class hierarchy. For example, classes extending `io.dropwizard.Configuration` received Spring Boot's `@ConfigurationProperties`, while `HealthCheck` implementations were converted to Spring's `HealthIndicator`. This approach proved most effective when a superclass definitively indicated the need for specific Spring Boot annotations.
2. **Annotation-Based Detection** We used existing annotations as migration triggers. Classes marked with `@Path` received Spring's `@Component` annotation to maintain their discoverability in the new framework. Similarly, we transformed security annotations like `@RolesAllowed` into equivalent `@PreAuthorize("hasRole(...)")` expressions.
3. **Direct Replacements** Some annotations required straightforward substitutions - for instance, replacing `@Auth` with `@AuthenticationPrincipal`. More complex cases, such as converting `javax.annotation.security.PermitAll` to `@PreAuthorize("permitAll()")`, required additional transformation logic or metadata.

Advanced Annotation Processing and Serialization

To handle complex annotation parameters, we implemented specialized Java visitors that enabled sophisticated value transformations. For example, when converting `@RolesAllowed("ADMIN")` to `@PreAuthorize("hasRole('ADMIN')")`, we needed to

account for Spring Security's role naming conventions. While this approach supported comprehensive annotation migrations with parameter transformations, it required custom Java classes rather than simple YAML declarations. To address OpenRewrite's serializability requirements, we developed classes implementing `SerializableFunction<T, R>`, allowing for complex transformations (like converting `"BASIC_GUY"` to `"ROLE_BASIC_GUY"`) while maintaining recipe serializability.

3.4.10 Cleanup

After completing the core migration steps, we apply a comprehensive cleanup recipe that addresses both general code quality and framework-specific *remnants*. The recipe combines standard static analysis tasks (like removing unused imports and fixing common anti-patterns) with specialized cleanup operations that remove lingering Dropwizard references. For framework cleanup, we systematically remove Dropwizard package references, unnecessary method overrides, and unused variables, while also shortening fully qualified type references for better readability. Although these automated cleanup steps handle most common cases, a final manual review ensures all framework migrations achieve their intended outcomes and verifies that the codebase remains functional after cleanup.

3.5 Conclusion

The development of the AST-based migration tooling using OpenRewrite, detailed throughout Section 3.4, culminated in a comprehensive and modular set of recipes designed to automate significant portions of the Dropwizard to Spring Boot transition. Our primary accomplishment was creating a flexible toolkit capable of handling complex refactorings specific to this framework change. This involved building a composite recipe structure (`ee.taltech.CompositeDropwizardToSpringBoot`) that orchestrated numerous custom and standard OpenRewrite recipes.

A core focus of our development effort centered on tackling challenging transformations frequently required in framework migrations. We invested significantly in creating robust recipes for handling complex *type changes* (Section 3.4.8), developing sophisticated *removal scripts* (Sections 3.4.2, 3.4.8, 3.4.10), and implementing *improvements in annotation processing* (Section 3.4.9). While developing these targeted solutions, our experience also highlighted recurring challenges inherent in this type of automated migration, including the difficulties in reliably handling diverse YAML configuration structures (Section 15), managing potential bean definition conflicts (Section 3.4.4), and fully automating complex test logic (Section 3.4.7).

Despite these inherent complexities, the AST-based approach enabled substantial automation of mechanical tasks. We addressed specific difficult cases pragmatically, for instance by recreating a compatible `AbstractDAO` base class (Section 3.4.6) or generating method stubs via recipe parameters (Section 3.4.8) where full logic translation was impractical. Crucially, the functionality and correctness of the developed recipes themselves were extensively validated through dedicated unit tests within the recipe module and by repeatedly running them against our target open-source and closed-source service repositories throughout the development cycle.

Based on this development experience, we can address **RQ1** concerning the effective design and structure of proof-of-concept recipes for complex cross-framework migrations with minimal intervention. Our findings suggest a **modular, composite structure** (Figure 7) is paramount Breaking down the migration into distinct functional areas (Configuration, Security, Data Access, Tests, etc.) managed by separate sub-recipes or modules allows for

customization, selective application, and easier maintenance and extension. This modularity enabled us to tailor the migration effectively for both the Example and Proprietary services by enabling or disabling specific modules.

Furthermore, effective recipe design should embrace **pragmatism and balance automation with strategic manual effort**, aligning with our *Path of Least Resistance* principle (Section 3.1.4). Rather than attempting perfect automation for highly complex or variable parts (like intricate test logic or parsing diverse YAML structures), recipes should focus on automating the repetitive, mechanical tasks reliably. For complex transformations, generating placeholders (like default properties, Section 3.4.3), skeleton files (like `SecurityConfig.java`, Figure 9), or method stubs (`addAbstractMethods`, Section 3.4.8) proved more cost-effective, providing developers with the correct structure while leaving the intricate logic for manual completion. Leveraging declarative YAML recipes for simpler tasks (like dependency management) and reserving custom Java visitors for complex AST manipulation (like advanced annotation processing) also optimizes development effort. Finally, designing recipes with **configurability** (e.g., parameterized versions, package names) enhances their reusability across different projects.

Recognizing the potential value of this work, the implementation and the core set of custom recipes reside in our open-source repository[2] (modules specific to proprietary components, like the `MigrateMetrics` module, are excluded). Furthermore, collaborating with the OpenRewrite maintainers, we initiated the dedicated `openrewrite/rewrite-dropwizard` module[3], where these scripts will be shared. This overview summarizes the development journey, emphasizing key technical accomplishments, challenges, and the derived recipe design principles, which underpins the overall migration results analyzed in Chapter 4.

[2]Recipe repository, Accessed: 2025-04-13, URL: `https://github.com/Fossur/openrewrite-dropwizard-springboot-recipe`

[3]OpenRewrite Dropwizard module, Accessed: 2025-04-13, URL: `https://github.com/openrewrite/rewrite-dropwizard`

4 Analysis

4.1 Methodology Overview

In this chapter, we analyze and compare the three distinct migration approaches detailed in Chapter 3 —Manual refactoring, LLM-assisted transformation (using OpenAI's o3-mini model), and AST-driven refactoring (via OpenRewrite recipes). To evaluate the effectiveness and trade-offs of each method across varying complexities, we applied these approaches to two different Dropwizard to Spring Boot migration scenarios. The structured data collection process, analysis tools, and performance metrics employed for this quantitative comparison are detailed below.

4.1.1 Data Collection Workflow

Separate Git branches were used for each migration approach (Manual, LLM Initial, LLM Finalized, AST Initial, AST Finalized). This branching strategy allowed the isolation of changes made during each phase and accurate tracking of the transformations applied during the *Initial* automated or manual pass versus the subsequent *Finalized* manual corrections and integration steps. Commits were made at each distinct stage to capture the state and facilitate analysis.

4.1.2 Tools

To gather high-level quantitative data about the code modifications, we utilized **Git Diff** [50]. This tool provided metrics such as the number of lines and files changed between the relevant commits established in our data collection workflow, offering a basic measure of the extent of changes introduced by each migration approach.

For more granular insights into the structural transformations applied to the code, we employed **RefactoringMiner** [51, 52, 53]. This tool allowed us to identify and count specific types of code refactorings performed between commits, yielding a higher-level understanding of the migration process beyond simple line counts. RefactoringMiner is

capable of programmatically detecting a substantial portion (40 out of 75 documented types [54, 55]) of standard refactorings.

4.1.3 Quantitative Comparison Metrics

To quantitatively compare the migration approaches, the analysis focused on the total effort required and a composite score reflecting overall effectiveness. The total effort is measured by **Time Spent (hrs)**, representing the total developer hours logged for each migration stage (*Initial, Finalized*).

Overall migration effectiveness is assessed using the **Overall Score**. This composite score combines the completeness achieved by the initial automated step (for the LLM and AST methods) with the total time efficiency, measured relative to the manual baseline. It is calculated in two stages: first determining the *Automation Completeness Score (ACS)*, which quantifies the proportion of the migration achieved automatically, as defined in Equation 4.1:

$$ACS = \frac{1}{3}\left(\frac{L_{Initial}}{L_{Total}} + \frac{D_{Initial}}{D_{Total}} + \frac{R_{Initial}}{R_{Total}}\right) \times 100 \tag{4.1}$$

In Equation 4.1, the three components provide a multi-faceted view of automated changes relative to the total effort required.

1. *Line Change Automation* ($L_{Initial}/L_{Total}$): This ratio quantifies the proportion of lines modified (added/deleted) in the *Initial* automated step compared to the total. While a straightforward measure of raw *volume* that can be skewed by formatting or large churn, it serves as a fundamental indicator of the *extent* of automated code intervention.
2. *Dependency Completeness* ($D_{Initial}/D_{Total}$): This ratio measures the comparison of dependencies handled automatically (*Initial* step) against the total dependency changes needed for the migration. This aspect is crucial for buildability and fundamental migration success, although the metric focuses on the presence/absence of dependencies rather than the complexity of their integration.
3. *Structural Automation* ($R_{Initial}/R_{Total}$): This ratio represents the comparison of structural refactorings detected by RefactoringMiner in the *Initial* phase versus the

total detected throughout the migration (*Initial* + *Finalized*). This offers insight into the *nature* and *complexity* of automated changes beyond simple line counts, though we acknowledge RefactoringMiner does not detect all refactoring types and detection does not guarantee functional correctness.

By averaging these three distinct facets with equal weight, ACS aims to provide a balanced perspective that mitigates the limitations inherent in any single metric, offering a useful comparative measure of the upfront automated contribution (in terms of lines, dependencies, and detected structures) across the different methods evaluated.

By definition, the Manual migration approach performs all work in the *initial* (and only) step, thus it is assigned an Automation Completeness Score of 100.

Next, we calculate a *Time Factor (TF)* for each approach (LLM, AST, Manual). This factor quantifies the time efficiency of a given migration method relative to the manual baseline, calculated according to Equation 4.2:

$$TF = \frac{T_{Manual}}{T_{Method}} \tag{4.2}$$

Where:

1. T_{Manual} is the total time spent on the Manual migration.
2. T_{Method} is the total time spent on the specific migration method being evaluated (Manual, LLM, or AST).

Note that for the Manual method, $T_{Method} = T_{Manual}$, resulting in a Time Factor of 1. Finally, the Overall Score is computed by combining the Automation Completeness Score (ACS) and the Time Factor (TF). This multiplication yields the final composite score, as shown in Equation 4.3:

$$\text{Overall Score} = ACS \times TF \tag{4.3}$$

This calculation yields a composite score where the Manual method serves as the baseline with an Overall Score of 100 (since $ACS = 100$ and $TF = 1$). Scores higher than 100

indicate methods that achieved a significant degree of automation ($ACS > 0$) in less time than the manual approach ($TF > 1$), effectively rewarding both automation completeness and time efficiency relative to the baseline. Conversely, scores below 100 might indicate methods that were either less complete in their initial automated pass or took longer than the manual refactoring.

4.2 Example Service Migration Results

The quantitative metrics in Table 3 clearly show the efficiency gains achieved through automation for the Example Service compared to the purely manual effort.

Metric	Manual	LLM	AST
Time Spent (hrs)	24	7.5	3.5
Line Changes Initial	1627	1572	1575
Line Changes Finalized	0	109	409
Line Changes Total	1627	1681	1984
Dependency Changes Initial	21	33	23
Dependency Changes Finalized	0	5	3
Dependency Changes Total	21	38	26
Structure Changes Initial	228	218	168
Structure Changes Finalized	0	2	132
Structure Changes Total	228	220	300
Automation Comp. Score (%)	100	93.15	74.6
Time Factor	1	3.2	6.86
Overall Score	100	298.1	511.7

Table 3. Example service migration metrics (showing *Initial* vs *Finalized* components)

LLM Approach Analysis

The *LLM approach* demonstrated significant benefits over manual work, achieving a high Overall Score of 298.1 in 7.5 hours (Table 3). The high score was primarily due to its excellent initial automation, reflected in a very high ACS of 93.15 (Table 3, Equation 4.1).

Evidence for this high ACS includes the minimal finalization needed: only 109 lines and 2 structural changes. Qualitatively, the LLM effectively handled boilerplate conversion and often rewrote syntax holistically towards an idiomatic Spring Boot style, performing particularly well on the smaller files of the Example Service.

While achieving good results in 7.5 hours, leading to a substantial TF of 3.2 (Table 3), this required considerable interactive developer effort. The recorded time included significant developer involvement in iterative prompting (estimated average 2-5 prompts per file), constant oversight, error correction (e.g., identifying missing annotations), and debugging generated outputs. This developer-intensive interaction model contrasts with the potentially more predictable, albeit numerous, finalization tasks associated with the AST method.

Regarding the build configuration, the LLM approach replaced existing Dropwizard dependencies with Spring Boot equivalent, but this adherence came at the cost of overreaching by removing existing, critical code quality plugins (such as Checkstyle and Jacoco) from the configuration. This necessitated subsequent manual effort to diagnose the omissions and re-integrate these essential tools, partially offsetting the automation gains achieved elsewhere.

AST Approach Analysis

The *AST approach* delivered the highest *Overall Score* (511.7, defined in Table 3). This outcome was predominantly driven by its exceptional speed, completing the migration in only 3.5 hours compared to 24 hours for the Manual baseline. This efficiency resulted in the highest TF of 6.86 (Table 3), calculated as defined in Equation 4.2. The speed advantage was partly attributed to effective automation of dependency setup, minimizing developer research time (Figure 5).

However, this speed came with a trade-off in initial completeness. The AST method achieved a moderate ACS of 74.6 (Table 3, definition in Equation 4.1). This is evidenced by the relatively large volume of changes requiring manual finalization compared to the LLM approach, specifically 409 lines and 132 structural changes. Even though many manual changes were required, they were finished quickly. This indicates that the automated recipes mostly left numerous small, targeted tasks—like implementing method stubs or adjusting security details—that could be completed rapidly by hand.

The AST-based migration for the open-source service effectively automated much of the dependency management. It successfully processed the majority of dependency modifications in the initial automated pass, including adding required Spring Boot libraries and removing obsolete Dropwizard ones. This efficient handling of dependencies was a key factor in reducing developer research time for the example service's migration.

A distinct advantage of the AST approach was its contribution to *reusable assets*. The effort invested resulted in validated OpenRewrite recipes (analyzed in Section 4.4), offering long-term value applicable to other migrations. This contrasts with the LLM's interactive session, which was specific to this single migration instance.

4.3 Proprietary Service Migration Results

The migration results for the more complex Proprietary Service (Table 4) show a stark divergence in the effectiveness of the automated approaches compared to both the Manual baseline and their performance on the simpler Example Service.

Metric	Manual	LLM	AST
Time Spent (hrs)	16	10	3
Line Changes Initial	670	1425	713
Line Changes Finalized	0	2359	351
Line Changes Total	670	3784	1064
Dependency Changes Initial	30	6	16
Dependency Changes Finalized	0	14	2
Dependency Changes Total	30	20	18
Structure Changes Initial	107	430	51
Structure Changes Finalized	0	33	65
Structure Changes Total	107	463	116
Automation Comp. Score (%)	100	53.5	66.6
Time Factor	1	1.6	5.33
Overall Score	100	85.6	355.3

Table 4. Proprietary service migration metrics (showing *Initial* vs *Finalized* components)

AST Approach Analysis

The *AST method* again achieved the highest Overall Score (355.3, Table 4). This was driven primarily by its exceptional speed (3 hrs) compared to the Manual baseline (16 hrs), resulting in a high Time Factor (TF) of 5.33 (Table 4, definition in Equation 4.2). Its Automation Completeness Score (ACS) remained reasonable at 66.6% (Table 4, definition in Equation 4.1), supported by high initial Dependency Completeness (approx. 89%), although requiring significant finalization for Lines (approx. 67% initial automation) and Structure (approx. 44% initial automation, requiring 65 finalized changes). This success in the proprietary context underscores the effectiveness of the *modular and targeted application of the recipes*.

As discussed in Section 4.4, we disabled some standard modules (Security, Hibernate, Tests) not necessary for this service and added a custom `MigrateMetrics` module (composed primarily of existing recipe types) alongside enhanced cleanup scripts. This tailoring allowed the AST approach to efficiently automate the applicable parts while bypassing proprietary complexities, aligning with the *Path of Least Resistance* principle.

LLM Approach Analysis

In stark contrast, the *LLM approach* struggled significantly with the Proprietary Service's specifics, resulting in an Overall Score of 85.6, substantially lower than the Manual baseline (Table 4). This low score quantitatively reflects the challenges encountered: the ACS was poor (53.5%, Table 4), particularly hindered by extremely low initial Dependency Completeness (only 30%).

Qualitatively, this aligns with the LLM's difficulties handling proprietary code: it suggested unsuitable public libraries for metrics, failed to manage internal dependencies (negating research time savings), and proposed conflicting exception handling patterns (`@ExceptionHandler` vs. custom logic), all requiring manual correction. Furthermore, the finalization metrics highlight a key failure mode. While the Structure Changes Finalized count was relatively low (33), indicating high initial structural automation (approx. 93%), the Line Changes Finalized count was extremely high (2359, Table 4). This large line difference primarily resulted from the complete manual reversion of a single large (~900 lines) resource file that the LLM failed to migrate correctly (as detailed in Section 3.3.2). This indicates that while the LLM might sometimes produce code appearing structurally plausible, its functional correctness can degrade severely on large, complex inputs, leading to significant line-level rework or wholesale reversion. It also underscores that the high count of finalized line changes here is skewed by this single revert.

Despite these significant issues limiting its overall effectiveness score, the LLM approach (10 hrs) was still faster than the purely manual one (16 hrs), achieving a Time Factor of 1.6 (Table 4).

4.4 Discussion

Applying the three migration methods across both the Example and Proprietary Service allowed us to synthesize overarching trends, identify context-specific variations, analyze the recipe development process, and draw conclusions addressing our research questions.

Consistently, a clear trade-off between development *time* and the resulting code *state* emerged. The *AST (OpenRewrite)* approach demonstrated the fastest execution times, achieving the highest Overall Scores (Tables 3 and 4), largely driven by significant time efficiency compared to the manual baseline. *LLM-assisted migration* substantially accelerated the process compared to manual efforts, serving as a valuable middle ground, although its effectiveness diminished notably with the proprietary complexities of the closed-source Service. *Manual migration*, while the most time-consuming, remained the benchmark for predictable quality and reliably handling undocumented logic.

Dependency handling also revealed method-specific strengths and weaknesses. Both the *AST* approach (when correctly configured with necessary recipes or exclusions) and the *Manual* method reliably achieved near-total Dependency Completeness relative to the finalized state in both scenarios. The *LLM*, conversely, showed variable reliability, performing adequately with standard public dependencies but struggling significantly with the proprietary libraries in the closed-source service, often requiring explicit guidance and manual correction.

The viability and effectiveness of the AST approach, central to this investigation, are intrinsically linked to the development and application of its recipe library. Our experience underscored that building these recipes requires a significant upfront investment. To analyze this systematically, we established a four-tier classification for recipe development effort:

Tier 0: Pre-existing recipes available in the OpenRewrite ecosystem that required no additional development time, only configuration and integration effort.

Tier 1: Simple transformation recipes that could be developed and tested within a single day, typically involving basic code structure changes.

Tier 2: Moderately complex recipes requiring one to three days of development time, often involving multiple related transformations or more sophisticated refactoring patterns.

Tier 3: Complex recipes taking over three days to develop - sophisticated syntax generation.

The distribution of the 63 distinct recipes used, depicted in Figure 20, shows that while nearly a fifth (19.7%) were pre-existing recipes (Tier 0) that could be leveraged, the majority of development time was concentrated on newly developed recipes. Specifically, recipes falling into Tier 2 (requiring 1-3 days each) constituted 52.5% of the total, and complex Tier 3 recipes (requiring over 3 days each) made up 11.5%. An additional 16.4% of the recipes were simpler Tier 1 custom creations (requiring up to 1 day). Evidently, complex cross-framework migrations demand significant custom recipe development, primarily to address moderately complex, recurring patterns (Tier 2) and intricate transformations (Tier 3).

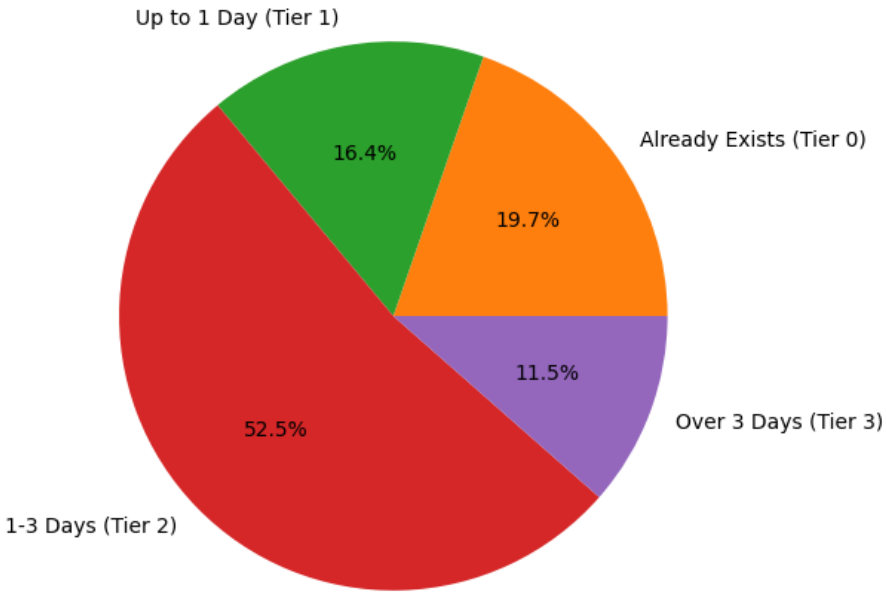

Figure 20. Recipe development effort distribution by tier

This concentration of effort in Tiers 2 and 3 stems from inherent challenges in automating complex code transformations via AST manipulation. For instance, maintaining type consistency across sequences of chained recipe applications proved difficult and error-prone, requiring careful design and validation within the recipe logic itself. Similarly, reliably converting longer or more complex method chains to an entirely different syntax (beyond simple annotation or type changes) demanded sophisticated, often brittle, recipe logic. Situations requiring complete method reimplementation presented a vast decision space

regarding functionally equivalent implementations, a task often ill-suited for deterministic AST rules that excel at structured, predictable transformations. Furthermore, significant code rearrangements—such as moving methods between classes or restructuring class hierarchies beyond simple parent changes—proved particularly challenging for reliable AST automation; these tasks are highly sensitive to the surrounding code context and architectural intent. Moreover, the *order* in which recipes are applied is critical. Recipes often build upon the code state (type information) created by previously executed ones, and incorrect sequencing can lead to unexpected failures or incomplete transformations, adding another layer of complexity to the migration process. These factors contribute significantly to the development time required for higher-tier recipes.

Metric	**Value**
Generally usable recipes	34
Created unique recipes	51
Total unique recipes used	63
Total recipes used in Open-Source service	105
Unique recipes used in Open-Source	61
Total recipes used in Closed-Source service	115
Unique recipes used in Closed-Source service	41

Table 5. High-level recipe metrics

Furthermore, this initial investment barrier is likely to decrease over time. As the open-source ecosystem around tools like OpenRewrite matures, with more pre-existing recipes becoming available and potentially covering more complex patterns, the effort required to assemble effective migration scripts for common framework transitions, such as the Dropwizard to Spring Boot path studied here, should diminish. This trend further improves the cost-benefit analysis for adopting the AST approach in the future.

The AST approach's modularity, demonstrated by the recipe library structure (Figure 3.4.1), proved crucial for its success on the Proprietary Service. By disabling inapplicable standard modules (`MigrateSecurity`, `MigrateHibernate (with Liquibase)`, `MigrateTests`) and composing a custom `MigrateMetrics` module largely from existing Tier 0 recipes, the method efficiently achieved the highest Overall Score (355.3) despite

the service's unique requirements. This showcases the approach's adaptability with low marginal effort through both module configuration and the reuse of foundational recipe components for specialized tasks (Table 4).

The LLM's limitations with complex, proprietary code were quantitatively evident in the closed-source service migration. Beyond the poor ACS (53.5%) and dependency issues, the extremely high count of finalized line changes (Table 4) — primarily due to reverting a single large (~900 lines), incorrectly migrated file (Section 3.3.2)—highlights a critical failure mode. This indicates that while the LLM might sometimes produce code appearing structurally plausible (high initial structural automation of approx. 93%), its functional correctness can degrade severely on large, complex inputs, leading to significant line-level rework or wholesale reversion. This contrasts sharply with its minimal finalization needs on the simpler Example Service (Tables 3). The developer-intensive interaction model required for the LLM (iterative prompting, oversight, debugging) also contrasts with the potentially more predictable finalization tasks associated with the AST method, even if those tasks are numerous.

Regarding the extent to which the Dropwizard to Spring Boot migration could be automated while preserving functional parity (**RQ2**), our findings indicate substantial but incomplete automation is achievable. The initial automated steps using AST and LLM successfully handled a substantial portion of the migration work, achieving significant automation coverage and translating directly into considerable time savings compared to the manual baseline (evidenced by Time Factors > 1, seen in Tables 3 and 4). However, the consistent need for non-zero finalization across both services and methods underscores that manual intervention remains essential. Handling complex business logic, nuanced configurations, comprehensive test adaptation, and ultimately ensuring functional correctness met validation criteria reliably required manual oversight, indicating that full, hands-off automation remains challenging for complex, real-world migrations.

4.5 Conclusion

In conclusion, this analysis confirms the AST-based approach, supported by a well-structured and tailored recipe library, offers superior execution speed and automation potential compared to LLM-assisted and Manual methods for this specific Dropwizard to Spring Boot migration context (**RQ3**). The experience highlighted both the significant upfront investment required for recipe development and the considerable rewards in efficiency derived from that effort. While limitations remain, particularly around automating highly dynamic code transformations, the targeted and deterministic nature of AST transformations, combined with modular configuration, proved highly effective. This was especially notable in navigating proprietary complexities where the generalized LLM approach faltered significantly. The recipe development effort yields reusable assets, enhancing the value proposition, particularly in scenarios involving multiple similar migrations. Ultimately, building upon the demonstrated effectiveness of the AST-based approach explored in this work, a blended strategy could further enhance *efficiency*, particularly concerning the upfront investment for complex migrations. This would involve utilizing AST recipes for their precision in core, strategic transformations and ensuring reliable application of recurring patterns, while leveraging LLMs specifically for the broader, more syntactically demanding transformations. Such a division of labor could directly address the significant upfront time and cost associated with developing complex (e.g., Tier 2 and Tier 3) AST recipes, by having the LLM handle the heavy lifting for intricate syntax conversions that are currently resource-intensive to automate via AST rules alone. This synergy has the potential to make the migration process faster and more cost-effective to initiate, while robust manual oversight and testing would, critically, remain essential for final validation and addressing nuanced domain logic.

5 Threats To Validity

Several factors may limit the broader applicability of this thesis's findings. The conclusions are drawn from migrating only two specific services: one open-source and one proprietary system. These services, while differing in complexity, might not fully represent the diversity of Dropwizard applications used in industry. Variations in application size, complexity, or architecture could influence the effectiveness and feasibility of the automation techniques explored. Furthermore, the study specifically targets the migration from Dropwizard 1.3 to Spring Boot 2.7.x. Organizations using different framework versions or Java versions might face unique challenges not covered in this research.

The generalizability of the LLM-assisted migration results is also influenced by the specific model and prompting strategy used. This research utilized OpenAI's o3-mini model with zero-shot prompts. Employing different large language models, specialized LLM tools, or more advanced prompting techniques could lead to substantially different outcomes, especially when working with proprietary codebases or intricate domain logic.

It's also important to recognize that multiple valid strategies exist for executing each migration step. For example, JAX-RS resources could be integrated using Spring Boot's Jersey support or rewritten entirely using Spring MVC. Likewise, differing choices regarding security configurations, transaction management, or data access strategies impact the specific migration recipes required and their automation potential. Therefore, the OpenRewrite recipes developed here represent one viable migration path, not an exhaustive solution applicable to all situations.

The AST-based recipes themselves, being a proof-of-concept, have practical limitations. Although designed for modularity, they do not encompass every possible Dropwizard feature, third-party bundle, or unique architectural element found in large-scale production systems. Services relying heavily on specialized integrations or less common Dropwizard features might need significant customization of the provided recipes.

Finally, external validity is constrained by the reliance on developer judgment and the necessity for manual finalization steps. This required manual effort, essential for handling complexities and verifying LLM outputs, inherently reduces the level of complete automation and introduces variability depending on the specific context and developer expertise. Consequently, the results should be interpreted with caution, underscoring the need for further validation across a wider range of services and organizational settings.

6 Future Work

While this thesis demonstrates the feasibility of automating Dropwizard to Spring Boot migrations, several research directions remain open. First, the AST-based approach could be extended beyond a single language boundary, such as exploring cross-language transformations within the broader JVM ecosystem. For instance, transforming Java-based Dropwizard services into Kotlin-based Spring Boot applications may present additional challenges and opportunities for advanced recipe design, particularly around language-specific syntactic constructs and idioms. Investigating these cross-language scenarios would help determine whether tools like OpenRewrite can effectively bridge linguistic gaps or if specialized frameworks are needed.

Future studies could also delve deeper into automating migrations with more comprehensive or specialized tooling. As LLMs continue to evolve, comparing different model providers, both open-source and commercial, could highlight variations in performance, context handling, or security/privacy trade-offs. Effective prompt-engineering techniques (such as few-shot or chain-of-thought prompting) and specialized frameworks for interactive code analysis might further refine or accelerate the migration process. More rigorous experiments with proprietary code and domain-specific libraries could shed light on the extent to which LLMs can generalize or require tailored context and examples.

An especially promising line of inquiry centers on a *combined approach* that unites AST- and LLM-based automation in a single pipeline. Here, LLMs could dynamically generate, refine, or customize OpenRewrite recipes by processing smaller chunks of code—allowing them to handle context more effectively and reducing the up-front recipe-development burden. The AST tooling would then execute these recipes at scale, retaining the precision and predictability of structured transformations while benefiting from the LLM's adaptability.

Larger-scale industrial case studies, involving multiple Dropwizard services of varying

complexities, would lend stronger evidence to the comparative advantages and limitations of each approach. Community-driven initiatives, such as dedicated OpenRewrite modules for specialized migrations or curated prompt libraries, could foster ongoing collaboration and promote the broader adoption of automated modernization techniques.

7 Conclusion

This thesis investigated the potential for automating the migration of legacy Java services from Dropwizard to Spring Boot. We employed Abstract Syntax Tree-based refactoring with OpenRewrite as the primary technique, and compared its efficacy against an LLM-assisted transformation and the traditional manual migration process. The study evaluated the efficacy, challenges, and trade-offs of each approach across two case studies: a moderately complex open-source service and a proprietary closed-source system.

Beyond these comparative findings, a significant practical contribution of this work is the development of a reusable library of OpenRewrite recipes tailored to the Dropwizard to Spring Boot migration. These core recipes have been released as an open-source project to benefit the wider community, lowering the barrier for other organizations facing similar modernization challenges and encouraging further community-driven refinement of the migration tools.

In conclusion, this study confirms that AST-based tooling (exemplified by OpenRewrite) offers a highly effective strategy for accelerating Dropwizard to Spring Boot migrations. It yields considerable time savings over manual migration and greater reliability than the current generation of LLM approaches, especially for complex, structured refactoring tasks where deterministic transformations are crucial. While the AST approach requires an upfront investment in creating and tuning migration recipes, this investment provides substantial returns, particularly in scenarios involving multiple similar migrations. Overall, the most successful modernization efforts will likely *combine* the strengths of all approaches: leveraging AST automation for core, repetitive refactoring tasks, utilizing LLMs for complex syntactic transformations, and relying on essential manual oversight and validation to ensure correctness and handle the nuanced complexities of real-world enterprise systems.

References

[1] Wesley K. G. Assunção et al. „Contemporary Software Modernization: Perspectives and Challenges to Deal with Legacy Systems“. In: *CoRR* abs/2407.04017 (2024). DOI: 10.48550/ARXIV.2407.04017. arXiv: 2407.04017. URL: https://doi.org/10.48550/arXiv.2407.04017.

[2] *Spring Framework 6: The Full Cost of Migrating from v5 to v6*. HeroDevs. Nov. 2024. URL: https://www.herodevs.com/blog-posts/spring-framework-6-the-full-cost-of-migrating-from-v5-to-v6 (visited on 03/20/2025).

[3] John Browne. *Comparing the cost of migrating to rewriting*. June 2016. URL: https://www.mobilize.net/blog/comparing-the-cost-of-rewrite-to-migration (visited on 03/20/2025).

[4] Renaud Pawlak et al. „SPOON: A library for implementing analyses and transformations of Java source code“. In: *Softw. Pract. Exp.* 46.9 (2016), pp. 1155–1179. DOI: 10.1002/SPE.2346. URL: https://doi.org/10.1002/spe.2346.

[5] Addo Zhang. *OpenRewrite Learning (Part 1): Basics and Principles*. Dec. 2024. URL: https://addozhang.medium.com/openrewrite-learning-part-1-basics-and-principles-642c8c9fe645 (visited on 03/20/2025).

[6] Aylton Almeida, Laerte Xavier, and Marco Túlio Valente. „Automatic Library Migration Using Large Language Models: First Results“. In: *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2024, Barcelona, Spain, October 24-25, 2024*. Ed. by Xavier Franch et al. ACM, 2024, pp. 427–433. DOI: 10.1145/3674805.3690746. URL: https://doi.org/10.1145/3674805.3690746.

[7] Olga Kundzich and Justine Gehring. *Generative AI for Automating Code Remediation at Scale*. July 2023. URL: https://www.moderne.ai/blog/generative-ai-for-automating-code-remediation-at-scale (visited on 03/20/2025).

[8] *Dropwizard Documentation*. Dropwizard Team. 2025. URL: https://www.dropwizard.io/en/latest/ (visited on 03/26/2025).

[9] *Dropwizard GitHub Repository*. Dropwizard Team and Contributors. 2025. URL: https://github.com/dropwizard/dropwizard (visited on 03/26/2025).

[10] *Spring Boot Project Page*. Spring. 2025. URL: https://spring.io/projects/spring-boot (visited on 03/26/2025).

[11] *Spring Boot GitHub Page*. Spring. 2025. URL: https://github.com/spring-projects/spring-boot (visited on 03/26/2025).

[12] Curtis Johnson. *Highlights From the 2024 Java Developer Productivity Report*. Mar. 2024. URL: https://www.jrebel.com/blog/2024-java-report-highlights (visited on 03/20/2025).

[13] Patricia Johnson. *Case study: Insurer improves developer productivity with code migration automation*. Jan. 2024. URL: https://www.moderne.ai/blog/case-study-improving-developer-productivity-with-code-migration-automation (visited on 03/20/2025).

[14] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. „Technical Debt: From Metaphor to Theory and Practice“. In: *IEEE Softw.* 29.6 (2012), pp. 18–21. DOI: 10.1109/MS.2012.167. URL: https://doi.org/10.1109/MS.2012.167.

[15] *Information Technology: Agencies Need to Develop and Implement Modernization Plans for Critical Legacy Systems*. GAO-19-471. U.S. Government Accountability Office, 2019. URL: https://www.gao.gov/products/gao-19-471.

[16] Jesus Bisbal et al. „Legacy Information Systems: Issues and Directions“. In: *IEEE Softw.* 16.5 (1999), pp. 103–111. DOI: 10.1109/52.795108. URL: https://doi.org/10.1109/52.795108.

[17] *Lift and Shift*. IBM. 2025. URL: https://www.ibm.com/topics/lift-and-shift (visited on 04/09/2025).

[18] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. „Migrating Towards Microservice Architectures: An Industrial Survey“. In: *IEEE International Conference on Software Architecture, ICSA 2018, Seattle, WA, USA, April 30 - May 4, 2018*. IEEE Computer Society, 2018, pp. 29–39. DOI: 10.1109/ICSA.2018.00012. URL: https://doi.org/10.1109/ICSA.2018.00012.

[19] Georg Buchgeher et al. „Adopting Microservices for Industrial Control Systems: A Five Step Migration Path“. In: *26th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2021, Vasteras, Sweden, September 7-10, 2021*. IEEE, 2021, pp. 1–8. DOI: 10.1109/ETFA45728.2021.9613622. URL: https://doi.org/10.1109/ETFA45728.2021.9613622.

[20] Janne Kauhanen. „Modernizing usability and development with microservices“. English. Master's Thesis. University of Helsinki, Faculty of Science, 2022. URL: http://hdl.handle.net/10138/352072.

[21] *Danske Bank Halves Large-Scale Migration Timeline with Hyperautomation on AWS*. Amazon Web Services. 2024. URL: https://aws.amazon.com/solutions/case-studies/danske-bank-hyperautomation-case-study/ (visited on 02/23/2025).

[22] *Codemod*. Archived repository. Facebook. 2024. URL: https://github.com/facebookarchive/codemod (visited on 11/27/2024).

[23] *jscodeshift*. Facebook. 2024. URL: https://github.com/facebook/jscodeshift (visited on 11/27/2024).

[24] Louis Wasserman. „Scalable, example-based refactorings with refaster“. In: *Proceedings of the 2013 ACM Workshop on Refactoring Tools, WRT@SPLASH 2013, Indianapolis, IN, USA, October 27, 2013*. Ed. by Emerson R. Murphy-Hill and Max Schäfer. ACM, 2013, pp. 25–28. DOI: 10.1145/2541348.2541355. URL: https://doi.org/10.1145/2541348.2541355.

[25] Rick Ossendrijver, Stephan Schroevers, and Clemens Grelck. „Automating Library Migrations with Error Prone and Refaster“. In: *SIGAPP Appl. Comput. Rev.* 23.1 (Apr. 2023), pp. 5–19. ISSN: 1559-6915. DOI: 10.1145/3594264.3594265. URL: https://doi.org/10.1145/3594264.3594265.

[26] *Refaster & ErrorProne*. Google. 2024. URL: https://github.com/google/Refaster (visited on 11/27/2024).

[27] *Refaster Recipes*. Moderne Inc. 2025. URL: https://docs.openrewrite.org/authoring-recipes/refaster-recipes (visited on 04/18/2025).

[28] *OpenRewrite*. Moderne Inc. 2024. URL: https://docs.openrewrite.org/ (visited on 11/27/2024).

[29] *What is the Lossless Semantic Tree (LST) code model for automated refactoring and analysis?* Moderne Inc. 2024. URL: https://www.moderne.ai/blog/lossless-semantic-tree-the-complete-code-data-model-for-automated-code-refactoring-and-analysis (visited on 11/27/2024).

[30] *OpenRewrite Recipes*. Moderne Inc. 2024. URL: https://docs.openrewrite.org/ (visited on 11/27/2024).

[31] *ChatGPT*. OpenAI. 2025. URL: https://openai.com/index/chatgpt/ (visited on 04/10/2025).

[32] *GitHub Copilot*. GitHub. 2025. URL: https://github.com/features/copilot (visited on 04/10/2025).

[33] Shraddha Barke, Michael B. James, and Nadia Polikarpova. „Grounded Copilot: How Programmers Interact with Code-Generating Models“. In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023), pp. 85–111. DOI: 10.1145/3586030. URL: https://doi.org/10.1145/3586030.

[34] Fang Liu et al. „Exploring and Evaluating Hallucinations in LLM-Powered Code Generation“. In: *CoRR* abs/2404.00971 (2024). DOI: 10.48550/ARXIV.2404.00971. arXiv: 2404.00971. URL: https://doi.org/10.48550/arXiv.2404.00971.

[35] Dong Huang et al. „Bias Assessment and Mitigation in LLM-based Code Generation“. In: *CoRR* abs/2309.14345 (2023). DOI: 10.48550/ARXIV.2309.14345. arXiv: 2309.14345. URL: https://doi.org/10.48550/arXiv.2309.14345.

[36] TechRadar. *Samsung Workers Leaked Company Secrets by Using ChatGPT*. Accessed: 2024-12-05. Apr. 2023. URL: https://www.techradar.com/news/samsung-workers-leaked-company-secrets-by-using-chatgpt.

[37] TechCrunch. *Samsung bans use of generative AI tools like ChatGPT after April internal data leak*. Accessed: 2024-12-05. May 2023. URL: https://techcrunch.com/2023/05/02/samsung-bans-use-of-generative-ai-tools-like-chatgpt-after-april-internal-data-leak/.

[38] *Leaderboards*. Aider. 2025. URL: https://aider.chat/docs/leaderboards/ (visited on 02/23/2025).

[39] Colin White et al. „LiveBench: A Challenging, Contamination-Free LLM Benchmark“. In: *CoRR* abs/2406.19314 (2024). DOI: 10.48550/ARXIV.2406.19314. arXiv: 2406.19314. URL: https://doi.org/10.48550/arXiv.2406.19314.

[40] *o3-mini*. OpenAI. 2025. URL: https://platform.openai.com/docs/models/o3-mini (visited on 04/10/2025).

[41] *Spring Boot Reference Documentation*. Spring. 2025. URL: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/ (visited on 03/26/2025).

[42] *The IoC container (Spring Framework 3.2.x)*. Spring. 2013. URL: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html (visited on 03/26/2025).

[43] B. Kanjarla. *SpringBoot vs DropWizard: A developer Point of view*. 2025. URL: https://medium.com/@bhargavkanjarla01/springboot-vs-dropwizard-a-developer-point-of-view-f67dad17c8d6 (visited on 03/26/2025).

[44] *Spring Cloud Documentation*. Spring. 2025. URL: https://spring.io/projects/spring-cloud (visited on 03/26/2025).

[45] *GraalVM Native Images*. Spring. 2025. URL: https://docs.spring.io/spring-boot/reference/packaging/native-image/index.html (visited on 03/26/2025).

[46] Shekhar Gulati. *My take on libraries over framework (Spring Boot vs Dropwizard)*. May 2022. URL: https://shekhargulati.com/2022/05/06/my-take-on-libraries-over-frameworkspring-boot-vs-dropwizard/ (visited on 03/26/2025).

[47] *Spring Runtime Support (VMware Tanzu)*. VMware. 2025. URL: https://www.vmware.com/products/app-platform/tanzu-spring (visited on 03/26/2025).

[48] *Spring Boot Reference Documentation (2.7.18)*. Spring. 2024. URL: https://docs.spring.io/spring-boot/docs/2.7.18/reference/html/getting-started.html (visited on 03/29/2025).

[49] *2024 State of the Java Ecosystem*. New Relic. 2024. URL: https://newrelic.com/resources/report/2024-state-of-the-java-ecosystem (visited on 03/29/2025).

[50] *git-diff Documentation*. The Git Project. 2025. URL: https://git-scm.com/docs/git-diff (visited on 04/07/2025).

[51] Nikolaos Tsantalis et al. „Accurate and Efficient Refactoring Detection in Commit History“. In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: ACM, 2018, pp. 483–494. ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180206. URL: http://doi.acm.org/10.1145/3180155.3180206.

[52] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. „RefactoringMiner 2.0“. In: *IEEE Transactions on Software Engineering* 48.3 (2022), pp. 930–950. DOI: 10.1109/TSE.2020.3007722.

[53] Pouria Alikhanifard and Nikolaos Tsantalis. „A Novel Refactoring and Semantic Aware Abstract Syntax Tree Differencing Tool and a Benchmark for Evaluating the Accuracy of Diff Tools“. In: *ACM Transactions on Software Engineering and Methodology* (Sept. 2024). Just Accepted. ISSN: 1049-331X. DOI: 10.1145/3696002. URL: https://doi.org/10.1145/3696002.

[54] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. 2nd. Addison-Wesley Professional, 2018.

[55] Martin Fowler. *Online catalog for Refactoring 2nd Edition*. Accessed: 2025-04-12. URL: https://refactoring.com/catalog/.

Appendix 1 – Non-exclusive Licence for Reproduction and Publication of a Graduation Thesis[1]

I Karl-Erik Hein

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Abstract Syntax Tree-Based Tooling For Java Framework Migration”, supervised by Gert Kanter
    1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
    1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright
2. I am aware that the author also retains the rights specified in clause 1 of the nonexclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

20.05.2025

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive licence shall not be valid for the period.