MigrationBench: Repository-Level Code Migration Benchmark from Java 8

Linbo Liu*
AWS AI
Santa Clara, CA, USA
linbol@amazon.com

Xinle Liu*
AWS AI
Santa Clara, CA, USA
sliuxl@amazon.com

Qiang Zhou
AWS AI
Sunnyvale, CA, USA
zhouqia@amazon.com

Lin Chen
AWS AI
Santa Clara, CA, USA
linchenk@amazon.com

Yihan Liu
AWS AI
Santa Clara, CA, USA
yyihanl@amazon.com

Hoan Nguyen
AWS AI
Seattle, WA, USA
hoanamzn@amazon.com

Behrooz Omidvar-Tehrani
AWS AI
Santa Clara, CA, USA
omidvart@amazon.com

Xi Shen
AWS AI
Santa Clara, CA, USA
xis@amazon.com

Jun Huan
AWS AI
Santa Clara, CA, USA
lukehuan@amazon.com

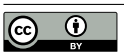
Omer Tripp
AWS AI
Sunnyvale, CA, USA
omertrip@amazon.com

Anoop Deoras
AWS AI
Santa Clara, CA, USA
adeoras@amazon.com

Abstract

With the rapid advancement of powerful large language models (LLMs) in recent years, a wide range of software engineering tasks can now be addressed using LLMs, significantly enhancing productivity and scalability. Numerous benchmark datasets have been developed to evaluate the coding capabilities of these models, while they primarily focus on code generation and issue-resolution tasks. In contrast, we introduce a new coding benchmark MigrationBench with a distinct focus: code migration. MigrationBench aims to serve as a comprehensive benchmark for migration from `Java 8` to the latest long-term support (LTS) versions (`Java 17, 21`), including a `full` dataset and its subset `selected` with 5,102 and 300 repositories respectively. `selected` is a representative subset curated for complexity and difficulty, offering a versatile resource to support research in the field of code migration. Additionally, we provide a comprehensive evaluation framework to facilitate rigorous and standardized assessment of LLMs on this challenging task. We further propose an agentic framework and demonstrate that LLMs can effectively tackle repository-level code migration to `Java 17`. For the `selected` subset with `Claude-4.5-Sonnet`, our agentic framework achieves 71.67% and 53.33% success rate (pass@1) for minimal and maximal migration respectively. The dataset and evaluation source code are available at: https://huggingface.co/collections/AmazonScience/migrationbench and https://github.com/amazon-science/MigrationBench respectively.

*Both authors contributed equally to this research.



KDD '26, Jeju Island, Republic of Korea

ACM ISBN 979-8-4007-2259-2/2026/08
https://doi.org/10.1145/3770855.3817526

CCS Concepts

• **Software and its engineering → Software maintenance tools**; *Empirical software validation*; • **Computing methodologies →** *Natural language generation.*

Keywords

Java 8 Repository Code Migration; Benchmark Dataset; Approximate Functional Equivalence; LLM Agents; Software Engineering

ACM Reference Format:
Linbo Liu, Xinle Liu, Qiang Zhou, Lin Chen, Yihan Liu, Hoan Nguyen, Behrooz Omidvar-Tehrani, Xi Shen, Jun Huan, Omer Tripp, and Anoop Deoras. 2026. MigrationBench: Repository-Level Code Migration Benchmark from Java 8. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD '26), August 09–13, 2026, Jeju Island, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3770855.3817526

1 Introduction

The rapid development of large language models (LLMs) with billions of parameters has led to an unprecedented growth in their size and reasoning capabilities [11, 20, 37]. The software engineering (SWE) industry involves various coding tasks with relatively objective goals [26], requiring strong domain knowledge and reasoning abilities. Consequently, the research community has been actively and effectively leveraging LLMs in this domain [7, 31]. Initially, LLMs were employed for standalone and interview-style code generation tasks [3, 6, 12]. For instance, GitHub Copilot[1] assists

[1] https://github.com/features/copilot

in completing functions within Integrated Development Environments (IDEs). Subsequently, their applications expanded to more complex use cases, like SWE-Bench [18], which generates pull requests to address open issues for GitHub repositories.

As LLMs demonstrated remarkable success in these coding tasks [15, 23, 33, 38], attention has shifted to even more complex challenges in software engineering, such as code migration [4, 28]. Unlike code generation or issue resolution, which often focuses on isolated functions or files, code migration tasks require a holistic approach involving the entire repository. Migration entails navigating a multi-step process to address numerous interconnected issues across files, making it a far more intricate task [21]. This complexity highlights the need to evaluate LLMs on such advanced use cases, especially considering the near-saturated performance on simpler benchmarks like code generation. However, creating a benchmark dataset for migration tasks poses significant challenges, given the breadth and depth of repository-wide changes involved.

To fill in this gap, we introduce MigrationBench, the first large-scale benchmark dataset in `Java`, designed specifically to assess LLM capabilities in code migration. Curating a high-quality migration dataset is a complex and multifaceted task that demands meticulous attention to various factors. Our approach employs a rigorous and scalable multi-step process that takes into account diverse aspects of the repository. We begin by applying license and repository quality filters to ensure compliance and maintain a certain level of quality. Subsequently, we employ robust techniques to identify the most appropriate snapshot as the starting point, ensuring the dataset's integrity and reliability. Furthermore, we design a comprehensive evaluation framework for migration success due to the absence of ground truth. Our approach ensures that both the benchmark datasets and final evaluation are of exceptional quality, providing researchers and practitioners with a reliable and comprehensive resource for studying and analyzing code migration patterns.

Our major contributions are summarized below:

(1) We introduce MigrationBench for repository-level code migration task from `Java 8`, *the* most popular programming language (PL) for around 20 years[2], the *first* large scale benchmark dataset among all PLs. The full MigrationBench dataset (`full`) contains 5,102 open-source Maven-based Java 8 repositories collected from GitHub. To accelerate future research on code migration, we further curate a more representative and challenging subset, `selected`, consisting of 300 repositories sampled from `full`. Both datasets are publicly hosted at https://huggingface.co/collections/AmazonScience.
(2) We propose an automated and comprehensive evaluation framework that verifies minimal migration requirements, *approximately* validates functional equivalence (FE) after migration, and checks whether dependencies are upgraded to their *latest* major versions in https://github.com/amazon-science/MigrationBench.
(3) We present an agentic workflow built on the Strands agent framework[3] to address code migration tasks. By providing basic shell access and migration knowledge base, the agent achieves reasonable migration efficacy. To further improve efficiency, we propose a hybrid approach that combines static code analysis with LLM-based agents. This hybrid method achieves performance comparable to the purely agentic approach while reducing LLM usage by 11%. We release source code at https://github.com/amazon-science/JavaMigration.
(4) Together with the source code, we also release all agent trajectories and their evaluation for `selected` subset to support reproducibility and facilitate future research. These trajectories provide fine-grained records of agent behavior, including intermediate decisions, tool invocations, and failure modes, which are critical for understanding the limitations of current agentic approaches. Beyond analysis, the released trajectories can also serve as valuable training data for developing and improving learning-based methods for code migration.

The paper is structured as follows: Section 2 provides an overview of related work in the field, situating our research within the existing body of knowledge. Section 3 details our meticulous process to curate the dataset. In Section 4, we delve into the evaluation metrics. Section 5 introduces our baseline solution with strands agent, an enhanced version by prompt engineering, a RAG-agent with migration knowledge base, and a hybrid approach combining static code analysis and LLM-based agents. Section 7 presents the experimental results obtained from applying our agentic solution to the curated dataset. Finally, we summarize major findings in Section 8.

2 Related Work

LLM Applications in the Software Engineering Industry. In the software engineering (SWE) industry, code maintenance [5] such as platform migration, library version upgrades, code refactoring and bug fixing, is an essential component in daily SWE work, requiring even significantly more SWE hours than developing new features [22]. Therefore, the community has an increasing interest in leveraging LLMs in nearly all fields of SWE tasks [10]. Reflexion [33], Agentless [38] and Agentcoder [15] mainly study the LLMs' ability in code generation to solve the interview-style coding problems. ChatUniTest [8] and TestPilot [31] explore the feasibility to use LLMs for automated unit test generation (UTG), while Toggle investigates fault localization and program repair [14].

Building on these foundational applications, the introduction of workflows and agents further enhances the utility of LLMs by addressing key limitations like hallucination and reasoning constraints through external feedback mechanisms [13, 15, 43]. Tools such as OpenDevin [35] and Cursor[4] excel at resolving complex coding tasks, automating repetitive processes and offering intelligent debugging solutions. Similarly, GitHub Copilot[5] and Amazon Q Developer[6] stand out for providing real-time code suggestions and enhancing developer productivity, while Replit Ghostwriter[7] focuses on collaborative coding and debugging. These innovations showcase how workflows and agents enable LLMs to handle longer, more complex trajectories effectively.

[2] https://www.tiobe.com/tiobe-index
[3] https://strandsagents.com
[4] https://www.cursor.com
[5] https://github.com/features/copilot
[6] https://aws.amazon.com/q/developer
[7] https://replit.com/learn/intro-to-ghostwriter

LLM Agents for Software Engineering. Recent research has increasingly investigated large language model (LLM)–based agents for solving software engineering (SWE) tasks, particularly in the context of automated bug fixing, code repair, and issue resolution. Many recent works propose agentic frameworks that decompose SWE tasks into multi-step decision processes, enabling LLMs to iteratively plan, invoke tools (e.g., search, execution, and testing), and incorporate feedback from the environment [33, 39, 41, 42, 47]. More recent methods explore search-based and reinforcement-learning–inspired strategies to guide action selection and refinement [19]. Despite these advances, there has been no systematic evaluation of LLMs on the code migration task. To address this gap, we propose an agentic framework for code migration on MigrationBench.

Benchmark Datasets: Code Generation, Maintenance and Translation. Benchmark datasets for coding tasks are abundant, but most focus on use cases other than code migration, as shown in a survey of around 200 coding benchmark datasets covering all phases of the software development life cycle comprehensively [34]. HumanEval [6] is a coding benchmark dataset designed to evaluate the correctness of LLMs in generating `Python` functions based on the provided docstrings. Mostly Basic Programming Problems (MBPP) [3] offers a more diverse dataset, assessing LLMs' overall problem-solving abilities in real-world coding scenarios with natural language problem descriptions. These datasets have been further enhanced into HumanEval$^+$ and MBPP$^+$ [25] to provide a more comprehensive evaluation framework for LLMs. APPS [12] extends the prior work by measuring the model's ability to take an arbitrary natural language specification and generate `Python` code. While the aforementioned benchmarks primarily focus on `Python` code, MBXP and Multilingual HumanEval [2] extend the evaluation to over 10 PLs, enabling a more comprehensive assessment of multilingual code generation capabilities.

DevAI [48], a benchmark of 55 realistic AI development tasks organized hierarchically as a directed acyclic graph (DAG), evaluates agentic systems on their ability to handle diverse aspects of AI development pipelines. In code maintenance, SWE-Bench [17] compiles a collection of popular `Python` repositories to facilitate research on leveraging LLMs for effective `Github` issue resolution. While SWE-Bench-Java [44] expands to the `Java` PL with 91 issues, SWE-PolyBench [30] is more comprehensive with 2,110 issues in multiple PLs, including `Java`, `JavaScript`, `TypeScript` and `Python`. Notably, while DevAI, SWE-Bench and its extensions include components that address broader development contexts with more advanced and difficult SWE tasks, none of them comprehensively tackles coding problems at the *repository* level as MigrationBench does.

Instead, repository-level research has been primarily focusing on code translation tasks, typically involving a few medium-sized repositories. Therefore, it lacks both a well-designed comprehensive benchmark dataset and sufficient complexity. While AlphaTrans [16] focuses on translating `Java` to `Python` for 10 repositories and RepoTransBench [36] attempts to translate 100 repositories in the opposite direction, neither of them has a ground truth target repository available. There are more research on translation to `Rust` as the target PL, from a source PL of `Go` [46] or `C` [32]. These translations only involve less than 10 repositories and up to 7,000 lines of code (LOC) for each repository. CodePlan [4] is closer to our work, demonstrating LLMs can resolve `C#` package migration and `Python` code edits, with 3 repositories for each use case. Several independent code migration benchmarks have been proposed *after* MigrationBench. CodeMenv [9] studies code migration at the *function* level, allowing up to three incompatible lines, and focuses on `Python` and `Java`. FreshBrew [27] emphasizes test coverage and includes 228 repositories, but adopts a less comprehensive curation and evaluation pipeline. In contrast, MigrationBench is a large-scale benchmark comprising more than 5,000 repositories and is accompanied by a robust, automated evaluation framework, making it well suited for systematic and reproducible evaluation of code migration methods.

3 MigrationBench

MigrationBench is a benchmark comprising a collection of `Github` open-source repositories, all written in a specific source language version (L_S). Given a target language version (L_T), the task is to migrate the repository to ensure compatibility with L_T with necessary code changes. In this paper, we present its `Java` version, that is L_S = `Java 8` and build tool is `maven`. These Maven-based applications are intended to be migrated to long-term support (LTS) L_T such as `Java 17` or 21. To facilitate large-scale repository processing, we leverage AWS Elastic Map Reduce (EMR) Serverless, running under `Amazon Linux 2` and `maven 3.9.6`, with 4 vCPUs and 16G memory for dataset curation.

3.1 Problem Formulation

Notations. Let R denote the content of a code repository including dependency declaration files (e.g. `pom.xml`), source code files, test files, auxiliary files (e.g. README, graphs, tables), etc. R is usually associated with a PL version L (e.g. `Java 8`, 17, 21) and can be built, tested or verified by a collection of verifiers $\mathbf{v}$. For example, a Maven `Java` repository R can be built and tested by $\mathbf{v}$ = `mvn clean verify`. We define a status function $f(R, L, \mathbf{v})$ that maps a code repository R, its associated language L and verifiers $\mathbf{v}$ to a boolean value, representing whether the verifiers pass or not, i.e. $f(R, L, \mathbf{v})$ = `True` or `False`.

Let R_S and R_T denote the code repository before and after the migration respectively. As MigrationBench does *not* aim at bug fixing, we require that the original code repository R_S is in a passing state before the migration under source language L_S, i.e. $f(R_S, L_S, \mathbf{v})$ = `True`. Given a target language L_T, we formulate the migration problem as to find a transformation from R_S to R_T such that $f(R_T, L_T, \mathbf{v})$ = `True`. Note that the ending states R_T subject to $f(R_T, L_T, \mathbf{v})$ = `True` are usually not unique.

In this paper we specifically discuss the scenario where L_S and L_T are `Java 8` and 17 respectively, and `mvn clean verify` is one component of the verifiers $\mathbf{v}$.

Note that some repositories may have hard-coded compiler versions in the dependency files (`pom.xml`), then they require *seed changes* (Δ_{seed}) [4] when migrating a repository. For example, a valid seed change can replace any hard-coded `Java 8` with 17 in `pom.xml` files.

3.2 Data Collection

We introduce the data collection process which produces the MigrationBench dataset. We scrape *all* Java repositories hosted in Github and apply multiple filters $\mathbf{F} \equiv \{F_i, 1 \le i \le 6\}$ sequentially detailed below to ensure high dataset quality.

F_1. Licenses. We collect Github repositories under the MIT or Apache 2.0 licenses only.

F_2. Enforce Repository Quality through Github Stars. To minimize the inclusion of potentially low-quality repositories, we require repositories to have at least 3 stars.

F_3. Build Tool is `maven`. For Java projects, Apache Maven, Gradle and legacy Apache Ant are all widely used build automation tools to manage dependencies, compile packages and run tests. In this paper, we limit the scope to maven, which is not only the most popular option for Java 8 but also represents a mature and well-documented build tool. As part of our criteria, a repository's *latest* snapshot must pass with the `mvn clean verify` command using at least one of Java 8, 11 or 17 without any modifications. This ensures that each selected repository is in a stable and passing state, so that it's reasonable to assume the existence of a valid migration path for those repositories.

By applying filtering up to $\mathbf{F_3}$, we narrow down to **16**, **154** repositories as of March 2024. All subsequent filters are conducted in March 2025.

Note. We encounter significant maven throttling when executing a high number of concurrent jobs, leading to potential *incompleteness* in the 16k list.

(1) Despite this, we consider false positives to be more detrimental than false negatives, thus the 16k list serves as a sufficiently robust and large-scale starting point for benchmark datasets like MigrationBench.
(2) Moreover, while false negatives may persist up to stage $\mathbf{F_3}$, we have adopted best-effort runtime strategies in subsequent stages, $\mathbf{F_4}$ through $\mathbf{F_6}$, to mitigate the throttling issue.

F_4. Search for L_S-compatible Base Commit ID (H_b). We check that the Github url remains valid, and then search for the last commit id, i.e. base commit H_b, which is compatible under source language L_S for each repository. L_S-compatibility for a repository's given snapshot is defined as below:

(1) It has at least one `pom.xml` file.
(2) It has valid source versions, i.e. each `pom.xml` file in a repository has either no hard-coded Java versions or the hard-coded version is Java 8, defined either through *maven.compiler.source*, *maven.compiler.target* and *maven.compiler.release* fields in *properties*, or through artifact *maven-compiler-plugin*'s *source*, *target* and *release* configurations.
(3) It's able to pass `mvn clean verify` with Java 8 and
(4) We further require the major versions of the final compiled classes, exported to the `target/classes/` or customized output directory, are 52. This class validation makes sure the *effective* `pom.xml` files are indeed Java 8.

Section A.1 covers more details on the assumptions and approximations for a faithful collection.

In the end, it reduces to **9**, **934** repositories with valid H_b and at least one `*.java` file present.

F_5. Dedup Repositories by Exact Match. There are numerous duplicate repositories on Github and $\mathbf{F_5}$ aims at removing them. Given a snapshot for a repository, we compute its *snapshot* id H^S for the whole repository at base commit H_b, by hashing the tree structure of the repository's files, *each* `*.java` file's hash and both filenames (relative to root) and raw content for *each* `pom.xml` file, concatenated by new lines. De-duplicating based on H^S results in **9**, **916** repositories and they'll be used as the starting point for MigrationBench.

F_6. Partition into `full` and `UTG` Subsets. As discussed in Section 7.2, the migration efficacy can be significantly different when including unit tests. Therefore, we further require repositories to have at least one test case to ensure at least partial functional equivalence after migration. We identify unit tests based on:

(1) Whether there are any `*.java` files present in the dedicated test directory `src/test/` or
(2) Whether there are any test cases triggered by `mvn test -f .` command in the root directory.

We further partition these repositories into a `full` dataset containing **5**, **102** repositories with unit tests, and a UTG subset containing the remaining **4**, **814** repositories without any unit tests. The remaining of this work primarily discusses `full` due to its direct relevance to migration evaluation, and we leave the UTG subset as a standalone benchmark for unit test generation.

3.3 `migration-bench-selected` (`selected`): A Representative and Challenging Subset

While we collect thousands of open-source repositories from Github to cover the *entire* problem space, the whole dataset is not always the best benchmark for evaluating LLMs' migration capabilities. A significant portion of the repositories may share similar issues, whereas a benchmark dataset should be diverse enough to reflect a range of challenges encountered in real-world migrations. Additionally, running LLM-assisted migration on all 5k repositories is prohibitively time- and resource-intensive.

We further introduce a smaller dataset to address these limitations, `migration-bench-selected` (`selected`). `selected` is a subset of **300** repositories carefully curated from `full` based on objective repository statistics and through human expert selection, which place greater emphasis on larger and multi-module repositories. By prioritizing repositories that present more complex migration challenges, `selected` is designed to better capture the diversity and intricacies of real-world code migration scenarios.

3.4 Dataset Statistics

We report statistics for both MigrationBench's `full` and `selected` subsets in Table 1. On average, repositories in the `full` subset have 2.4 modules, 7, 698.3 cumulative LOC, 65.1 `java` files, 8.7 test files in the `src/test` directory and 34.8 test cases. For the `selected` subset, metrics' average, median and standard deviation values are typically larger than the `full` superset.

4 Evaluation

Measuring the success of typical code generation or SWE benchmarking tasks is usually well-defined, as either the ground truth

Table 1: MigrationBench statistics at `base commit` H_b for both the `full` and the `selected` subset. For rows reporting average values, the highest and second-highest maximal values are highlighted in bold and underlined, respectively.

Metric[†]	Statistics	`full`	`selected`		
			all	success[‡]	failure[‡]
Size	-	5,102	300	46	254
#Modules	avg	2.4	<u>6.6</u>	2.4	**7.4**
	std	5.8	**15.1**	-	-
	median	1	1	-	-
	max	192	192	-	-
LOC	avg	7,698.3	<u>22,397.2</u>	**23,771.2**	22,148.4
	std	35,937.3	**108,203.9**	-	-
	median	2,044.5	**7,351.5**	-	-
	max	1,773,940	1,773,940	-	-
#Java Files	avg	65.1	<u>168.4</u>	123.4	**176.5**
	std	218.6	**650.7**	-	-
	median	25	**82**	-	-
	max	10,970	10,970	-	-
#Test Files[§]	avg	8.7	<u>8.8</u>	**9.3**	8.7
	std	**26.6**	21.8	-	-
	median	**2**	1	-	-
	max	**815**	194	-	-
#Test Cases[★]	avg	34.8	<u>110.6</u>	**354.9**	70.0
	std	289.9	**854.1**	-	-
	median	2	**8.5**	-	-
	max	14,450	14,450	-	-

[†] Other than the Size metric, all other metrics are computed on a per-repository basis.
[‡] We partition the whole `selected` dataset (`all`) into `success` and `failure` subsets, based on the maximal migration result of baseline Strands agent with `Claude-4.5-Sonnet`.
[§] It counts all `*.java` files in the `src/test/` directory only.
[★] It counts test cases by executing `mvn test -f .` in the root directory.

solution is available or it's straightforward to conduct an evaluation with comprehensive tests focusing on a limited scope e.g. with only *function* level or *file* level changes. Thus, they have very limited room for variations though the solutions are not necessarily unique. However, none of them applies for coding problems at the *repository* level. Furthermore, code migration evaluation presents unique challenges in that necessary code changes to L_T typically scale with repository size, which may vary significantly depending on the application types.

Approximate Functional Equivalence. The most rigorous way to evaluate success is to verify FE before and after migration, which is challenging by nature due to the huge problem and solution space. Instead, we adopt a few key metrics as approximations below, primarily replying on `mvn clean verify` or `compile` command and a few **invariants** as the measure of migration success, to automate the final evaluation and enable comparison across various migration efforts.

4.1 Minimal Migration

Maven Command Success (r_1). For a successful migration to `Java 17`, the minimum requirement is that the application builds successfully and passes all existing unit and integration tests when running on `Java 17`, verified by the `mvn clean verify` command. It performs a series of steps: it cleans cached files, compiles the source code, runs unit tests, packages the application, executes integration tests and validates the entire build process. When it completes without any build or test errors, the build is considered successful.

Compiled Class Major Version Validation (r_2). To ensure that the compiler is genuinely using `Java 17`, we implement a guardrail to verify the compiled classes' major versions are indeed 61.[8] These compiled classes are stored in `*/target/classes/**/*.class` usually.

Invariance: List of Test Methods (r_3). To make sure test methods (annotated with `@Test`) are not renamed, disabled or even completely removed after code migration, we compare test classes and annotated test methods based on abstract syntax tree (AST) parsing for all files in the dedicated `src/test` directory before and after migration, and the migration is considered unsuccessful if the test method names don't match.

[8] https://mkyong.com/java/list-of-java-class-file-major-version-numbers/

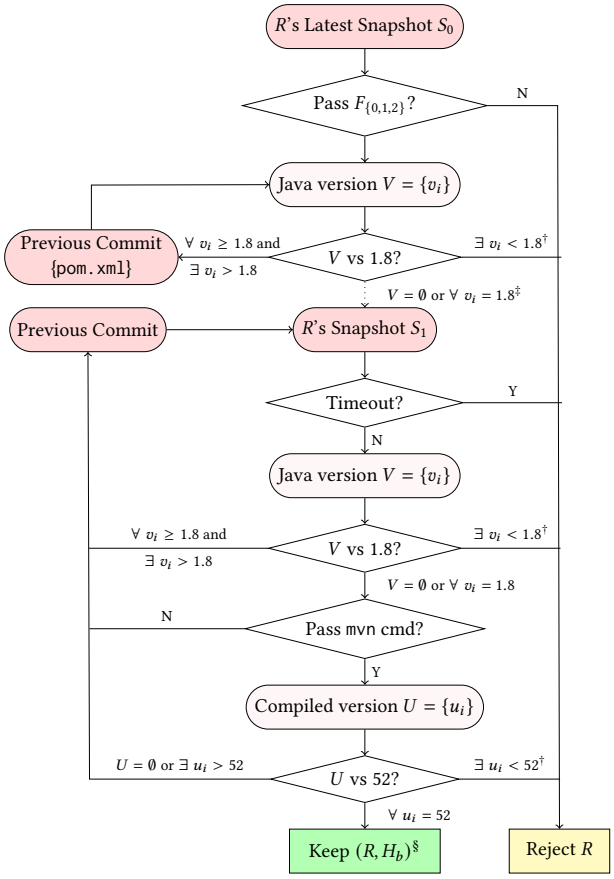


Figure 1: The flowchart to find out the base commit id H_b for code migration, starting with a `Github` repository R's latest snapshot as of March 2024.

† It assumes both the *explicitly* hard coded `java` versions and complied class major versions are monotonic over time.
‡ Given the commit history of V based on all `pom.xml` files, one can infer the first candidate snapshot S_1.
§ H_b is the commit id of the *last* S_1 where java version V passes version check, `mvn` command succeeds and compiled class major version U passes version check.

Invariance: Number of Test Cases (r_4). Static analysis of test directory in r_3 still does **not** guarantee runtime behaviors, therefore we further require number of test cases to be non-decreasing, primarily to avoid disabling those tests in `pom.xml` files hence effectively treating them as plain text files.

When a migrated repository satisfies requirements $\{r_1, r_2, r_3, r_4\}$, we define it as **minimal migration** and the corresponding efficacy (`pass@1`) is:

$$\eta_{\text{minimal}} = \frac{\#\text{ migrated repositories passing } \{r_1, r_2, r_3, r_4\}}{\#\text{ total repositories}} \tag{1}$$

4.2 Maximal Migration

Dependencies' *Latest* Major Versions (r_5). One key advantage of upgrading `Java` is to leverage the dependency libraries' latest major versions for efficiency and security reasons. For instance, the latest Spring Boot version 3.3.x is incompatible with `Java` 8, requiring at least `Java 17` since 3.0.[9] Therefore, to ensure a modern and robust migration, we require using **all** dependencies' stable and *latest* major versions available on Maven Central as of November 2024. We specify a list of version requirements for the most frequent 240 dependencies occurring in the `selected` subset. For the complete list of dependency versions, check https://github.com/amazon-science/MigrationBench/blob/main/src/migration_bench/reference/dependency_version.json.

Similar to minimal migration, we define **maximal migration** and the corresponding efficacy is:

$$\eta_{\text{maximal}} = \frac{\#\text{ migrated repositories passing } \{r_1, r_2, r_3, r_4, \mathbf{r_5}\}}{\#\text{ total repositories}} \tag{2}$$

Note that (i) By definition $\eta_{\text{maximal}} \leq \eta_{\text{minimal}}$ and (ii) We leverage `mvn dependency:tree` command to retrieve the effective version for each dependency, instead of simply relying on the dependency declaration files (`pom.xml`)

Compilation Efficacy. We also define $\eta^{\text{compile}}_{\text{minimal}}$ and $\eta^{\text{compile}}_{\text{maximal}}$, by dropping $\{r_3, r_4\}$ and replacing r_1 with r_1', a *weaker* command `mvn clean compile`, to address compilation errors only.[10]

Complementary to Sections 4.1 and 4.2, there could be even more specifications on migration tasks, and an incomplete list includes a more comprehensive FE check through enhancing test coverage in Section C.1, adopting modernized syntax and new security features in L_T, etc.

5 Agentic Framework

5.1 Strands Agent: Baseline

Large Language Model (LLM) agents have recently demonstrated strong capabilities in solving software engineering (SWE) tasks, including repository-level code understanding, modification, and validation. Prior work such as [41] and [39] shows that LLM-driven agents can autonomously perform non-trivial development workflows by iteratively reasoning, editing code, and executing commands. These results motivate the use of agentic systems for large-scale Java repository migration tasks.

We adopt the Strands agents framework as the backbone of our agentic design. Strands agent is a simple yet powerful SDK for building and running AI agents using a model-driven approach. It supports a wide spectrum of use cases—from lightweight conversational assistants to complex autonomous workflows—and scales seamlessly from local development to production deployment. As a baseline, we equip a Strands agent with a shell tool and an edit tool, enabling it to inspect the repository, modify source files, and execute build commands. The agent is instructed to ensure that `mvn clean verify` completes successfully without errors under `Java 17`. This configuration allows the agent to perform basic Java version migration and resolve straightforward compilation or test failures.

5.2 PE Agent (Baseline + PE)

While effective for minimal migration, the standalone Strands agent does not effectively conduct *maximal migration*, where all dependencies are upgraded to their latest major versions as shown in

[9] https://docs.spring.io/spring-boot/docs/3.0.0/reference/html/getting-started.html
[10] https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle

Table 2. To address this limitation, we further augment the agent's *system* prompt with explicit instructions to update each dependency to its latest major release whenever possible. This variant relies purely on prompt engineering (PE) on top of the baseline agent and serves as a stronger baseline for maximal modernization.

5.3 RAG Agent (Baseline + PE + RAG)

However, without external knowledge of current dependency versions, the agent may fail to identify the correct latest releases, especially for fast-evolving or less common libraries. To overcome this, we construct a dependency knowledge base and enable retrieval-augmented generation (RAG). In this setting, the agent can query the knowledge base to retrieve authoritative information about the latest major versions of dependencies before performing updates. We refer to this configuration as the *RAG-agent*.

5.4 Hybrid Approach

RAG-agent consumes additional tokens when searching from the knowledge base. To further reduce agent execution cost and improve efficiency, we propose a hybrid approach that combines static code analysis with agentic reasoning. In this design, a static analysis tool first identifies and upgrades all dependencies to their latest major versions in a deterministic manner. The PE agent is then invoked to fix resulting build failures, resolve incompatibilities, and complete any remaining migration steps. This hybrid approach leverages the strengths of both domain knowledge injection through automated code analysis tooling and LLM-based agents, achieving effective maximal migration with lower overall cost. Check Section 6 for detailed prompt templates.

6 Agent Prompt Templates

We provide additional details of the agent implementation introduced in Section 5. For the baseline agent, we grant access to a shell tool and an edit tool, enabling it to inspect, modify, and execute code. To restrict the agent's access to the target repository, the shell tool is sandboxed to operate only within the given project directory. We set the maximum number of agent turns to 80 to balance effectiveness and cost.

6.1 System Prompt

We show agent specific system prompts in this section.

6.1.1 Strands Agent: Baseline. Here is the system prompt for the baseline Strands agent:

System Prompt for the Strands Agent: Baseline

```
You are an expert Java developer assistant who can
migrate Java projects from JDK 8 to JDK 17. Make sure
`mvn clean verify` pass with JDK 17 after migration.
When `mvn clean verify` succeeds, you can conclude the
task. You don't have to provide any summary.
```

6.1.2 PE Agent. To emphasize maximal migration, for the PE agent we augment the system prompt with additional instructions requiring all dependencies to be upgraded to their latest major versions. This prompt engineering explicitly guides the agent to prioritize comprehensive dependency modernization beyond minimal build success:

System Prompt for the PE Agent

```
You are an expert Java developer assistant who can
migrate Java projects from JDK 8 to JDK 17. Make sure
`mvn clean verify` pass with JDK 17 after migration.
You should update all dependencies in the `pom.xml`
file to their latest versions that support Java 17.
When `mvn clean verify` succeeds, you can conclude
the task. You don't have to provide any summary.
```

6.1.3 RAG Agent. In some cases, the PE agent struggles to identify the latest major versions of dependencies using its internal knowledge alone, particularly when those versions were released after the LLM's knowledge cutoff. To address this limitation, the RAG agent leverages a search tool to retrieve up-to-date dependency information from an external knowledge base:

System Prompt for the RAG Agent

```
You are an expert Java developer assistant who can
migrate Java projects from JDK 8 to JDK 17.
Make sure `mvn clean verify` pass with JDK 17 after
migration.

You have access to a dependency version lookup
tool. When updating dependencies in pom.xml:
1. Use the search_dependency_version tool to look up
the recommended Java 17 compatible version for each
dependency
2. If a dependency is not found in the database, use
your knowledge to select an appropriate version
3. Update all dependencies to their Java 17 compatible
versions
```

6.1.4 Hybrid Approach. To further improve migration efficiency, the hybrid approach combines static code analysis with an agentic method. Specifically, a static parser first upgrades all dependencies to their latest major versions using version information stored in our knowledge base. While this step is deterministic and cost-effective, it may introduce build or test failures due to breaking changes. We therefore invoke the agent afterward to diagnose and fix the resulting errors and to complete any remaining migration steps. The system prompt for the hybrid approach is shown below:

System Prompt for the Hybrid Approach

```
You are an expert Java developer assistant who can
migrate Java projects from JDK 8 to JDK 17. Make sure
`mvn clean verify` pass with JDK 17 after migration.
Dependencies in the `pom.xml` file have been updated
to their latest versions that support Java 17, but
these changes might introduce compatibility issues
in the codebase. Please fix any such issues in your
migration. Do not downgrade the dependency versions
back to their JDK 8 compatible versions.
```

6.2 Initial User Prompt

For all four agent variants, we use the same initial user prompt as the input to ensure a controlled and fair comparison.

Initial User Prompt for All Agents

```
The code repository located at {repo_path} is currently written in Java 8. Please migrate the entire codebase to Java 17.
```

7 Experiments

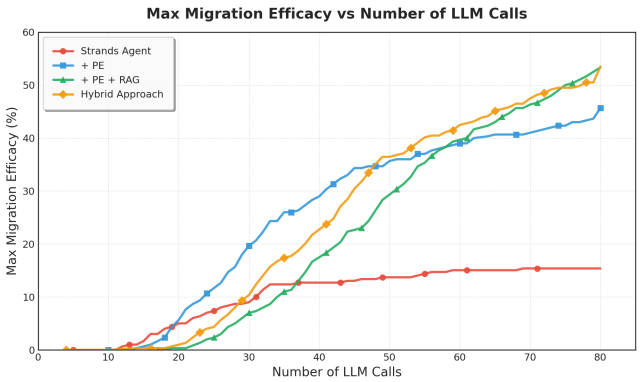


Figure 2: Maximal migration efficacy increases with more LLM calls for the following methods: Strands Agent (standalone baseline) with prompt engineering (+ PE) and RAG (+ PE + RAG), and Hybrid Approach.

We conduct multiple experiments to collect efficacy metrics as reference points for future research under `selected` subset. In addition to the Strands agents and hybrid approach introduced in Section 5, we also consider a static baseline using OpenRewrite recipe `org.openrewrite.java.migrate.UpgradeToJava17`[11], which deterministically upgrades Java applications to `Java 17` without involving generative AI.

For all agentic methods, we use the Strands agent framework with `Claude-4.5-Sonnet`(thinking-mode enabled) [1] as the underlying LLM, served via AWS Bedrock. This setup ensures a consistent agent architecture and model provider across experiments. We release all agent trajectories and their evaluation for `selected` subset to support reproducibility and facilitate future research.

We also discuss efficacy under two verification criteria, `mvn clean verify` and `mvn clean compile` to explicitly highlight the role of unit tests in code migration. While `mvn clean compile` captures basic compilation success, `mvn clean verify` provides a more strict evaluation by validating test correctness, thereby offering a more comprehensive measure of real-world migration quality.

7.1 Results

As discussed in Section 4.2, maximal migration requires more than simply building the repository without errors, which is sufficient only for minimal migration. Therefore, we primarily focus on analyzing maximal migration efficacy, as it better reflects real-world migration quality [29]. For completeness, we also report and discuss selected results on minimal migration.

7.1.1 Efficacy. In Table 2, we report minimal and maximal migration efficacy, with a cutoff of 80 LLM API calls (equivalent to 80 turns with the Strands agent) for agentic methods. Since the PE agent, RAG agent and the hybrid approach aim at solving maximal migration problem, they are not evaluated on minimal migration. We also plot efficacy dependence on the number of LLM API calls in Figure 2, emphasizing that efficacy is *not* a static number, and it depends on inference-time compute.

7.1.2 Open-source models. We also present maximal migration efficacy report for open source models such as `DeepSeek-V3.1` [24], `Qwen3-Coder-480B` [40] and `GLM-5` [45]. We run the hybrid approach on `migration-bench-selected` with the above models accessed via Amazon Bedrock. We report the migration efficacy in Table 3.

7.2 Discussion

7.2.1 Minimal and Maximal Migration Efficacy. Table 2 clearly shows a non-trivial gap between η_{minimal} and η_{maximal}. For Strands agent with `Claude-4.5-Sonnet` model, η_{minimal} is 4.7× of η_{maximal}. It implies that though minimal migration is relatively straightforward, maximal migration is much more challenging, since dependency packages may have to go through major version upgrades and are likely to introduce deprecated or breaking APIs with the same dependency, or even deprecated or completely renamed dependencies.

Comparative analysis based on the Claude model is shown in Table 1, demonstrating that maximal migration is more challenging for larger repositories in terms of number of modules and number of Java files.

7.2.2 Efficiency Study of $\eta_{maximal}$. We study the efficiency of the different approaches introduced in Section 5 by jointly analyzing maximal migration efficacy and the associated # LLM calls. Since the PE agent, RAG agent, and the hybrid approach are explicitly designed to solve the maximal migration problem, they are evaluated only under the maximal migration setting in Table 2.

Table 2 reports maximal migration efficacy together with the average number of LLM calls per repository. The purely static OpenRewrite baseline incurs zero LLM cost but achieves only 2.00% maximal migration efficacy, indicating its limited ability to handle complex migration issues. The baseline Strands agent improves maximal efficacy to 15.33% at an average cost of 33.68 LLM calls per repository. Adding prompt engineering substantially boosts maximal efficacy to 45.67%, albeit with increased LLM usage. Incorporating RAG further improves maximal efficacy to 53.33%, demonstrating the importance of external dependency knowledge for solving advanced migration failures. The hybrid method achieves the same maximal efficacy (53.33%) as the RAG agent with less LLM calls, highlighting effective cost reduction when combining deterministic tooling with agentic reasoning.

Figure 2 illustrates maximal migration efficacy as a function of the number of LLM calls. The baseline Strands agent quickly

[11] https://docs.openrewrite.org/recipes/java/migrate/upgradetojava17

Table 2: Maximal migration efficacy for `migration-bench-selected`. Since PE/ RAG agents and the hybrid approach aim at solving maximal migration problem, they are not evaluated on minimal migration.

Method	Type	η_{maximal} (%) ↑	η_{minimal} (%) ↑	Avg. # LLM Calls per Repo ↓
OpenRewrite	Static	2.00	16.33	-
Strands Agent	Agentic	15.33	**71.67**	**33.68**
+ PE	Agentic	45.67	-	49.22
+ PE + RAG	Agentic	**53.33**	-	59.22
Hybrid Approach	Hybrid	**53.33**	-	52.55

Table 3: Maximal migration efficacy for `migration-bench-selected` with hybrid approach and different models.

Models	η_{maximal} (%)	Avg. # LLM Calls per Repo
DeepSeek-V3.1	6.33	**45.93**
Qwen3-Coder-480B	22.33	50.27
GLM-5	45.33	47.20
Claude-4.5-Sonnet	**53.33**	52.55

plateaus at low efficacy, indicating diminishing returns from additional interaction steps. Prompt engineering significantly improves early-stage efficacy, achieving strong gains with relatively fewer calls. RAG and the hybrid approach both exhibit more gradual but sustained improvements, ultimately reaching the highest efficacy levels. Notably, the hybrid method achieves competitive efficacy with fewer agent-driven reasoning steps in early stages, reflecting the benefit of offloading dependency upgrades to static analysis before invoking the agent.

Overall, these results reveal a clear trade-off between migration quality and LLM cost. While richer agentic capabilities lead to higher maximal migration efficacy, the hybrid approach offers a favorable balance by achieving state-of-the-art efficacy with controlled LLM usage, making it a practical and cost-effective solution for large-scale Java migration.

7.2.3 Compilation Efficacy. Recall that we remove all repositories without test cases from `migration-bench-full`, as described in $\mathbf{F_6}$ of Section 3.2. This design choice is motivated by the critical role of unit tests in ensuring functional equivalence before and after migration.

To further support this claim, we conduct an ablation study in which all unit tests are disabled to examine whether migration efficacy increases. Concretely, we replace the default Maven command in r_1 with `mvn clean compile`, which only checks whether the source code compiles successfully and does not evaluate test correctness. Under this relaxed criterion, the minimal migration efficacy increases substantially, from **71.67%** to **97.67%**.

8 Conclusion

We introduce MigrationBench, a comprehensive code migration benchmark dataset from `Java 8` to subsequent LTS versions like `Java 17` and 21, with both a `full` dataset and its `selected` subset containing 5, 102 and 300 repositories respectively, reflecting real-world code migration scenarios. We design a comprehensive evaluation framework to approximate functional equivalence and rigorously assess migration success. This framework encompasses key criteria, including `maven` validations, verification of compiled class major versions, consistency in the list of test methods, non-decrease in the number of test cases and adherence to optional dependency version requirements (particularly relevant in maximal migration scenarios). We demonstrate the feasibility of code migration from `Java 8` to 17 through an agentic workflow with a few variations of the Strands agents (standalone, + PE, + PE + RAG, and a more cost-effective hybrid approach), and show preliminary results with promising efficacy for both minimal (71.67%) and maximal (53.33%) migration for the `selected` subset with `Claude-4.5-Sonnet`. We envision MigrationBench as a valuable resource for LLM practitioners and researchers, fostering further exploration and innovation in coding tasks.

9 Limitations

We summarize a few limitations of the current work below.

Test Coverage. We don't enforce test coverage and fully reply on *existing* test cases, although we do require the existence of test cases or dedicated test directory in source language L_S. Functional equivalence has a stronger guarantee when both the source code and tests are migrated to and pass under the target language L_T. However, for lines without any coverage, `maven`'s `verify` command is equivalent to `compile` only. Fortunately, our `UTG` subset can fill in the gap, serving as a benchmark for unit test generation. We leave unit test generation with `UTG` subset as a promising future research direction.

Throttling with `mvn`. We *completely* rely on running verifiers (e.g. various `maven` commands) when searching for the base commit ID H_b, running agentic framework and conducting the final evaluation. However, we encounter severe throttling when there are too many concurrent requests to maven central, therefore it's not rare to introduce false negatives and reproducibility issues. While this is not a major issue with the `selected` subset, it remains a challenge when searching for H_b to get the `full` subset with AWS EMR service, and we might end up with a base commit other than the actual H_b or completely miss a repository.

References

[1] Anthropic. System card: Claude sonnet 4.5, 2025. Accessed: 2026-02-02.

[2] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*, 2022.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[4] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):675–698, 2024.

[5] Gerardo Canfora and Aniello Cimitile. Software maintenance. In *Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals*, pages 91–120. World Scientific, 2001.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[7] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

[8] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation, 2024.

[9] Keyuan Cheng, Xudong Shen, Yihao Yang, TengyueWang TengyueWang, Yang Cao, Muhammad Asif Ali, Hanbin Wang, Lijie Hu, and Di Wang. CODEMENV: Benchmarking large language models on code migration. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of the Association for Computational Linguistics: ACL 2025*, pages 2719–2744, Vienna, Austria, July 2025. Association for Computational Linguistics.

[10] Hao Ding, Ziwei Fan, Ingo Guehring, Gaurav Gupta, Wooseok Ha, Jun Huan, Linbo Liu, Behrooz Omidvar-Tehrani, Shiqi Wang, and Hao Zhou. Reasoning and planning with large language models in code development. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 6480–6490, 2024.

[11] Deep Ganguli, Danny Hernandez, Liane Lovitt, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova Dassarma, Dawn Drain, Nelson Elhage, et al. Predictability and surprise in large generative models. In *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency*, pages 1747–1764, 2022.

[12] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

[13] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[14] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. A deep dive into large language models for automated bug localization and repair. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.

[15] Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

[16] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. Repository-level compositional code translation and validation, 2024.

[17] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024.

[18] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[19] Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

[20] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[21] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Müller, and J. Mylopoulos. Code migration through transformations: an experience report. In *CASCON First Decade High Impact Papers*, CASCON '10, page 201–213, USA, 2010. IBM Corp.

[22] Bennet P Lientz, E. Burton Swanson, and Gail E Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

[23] Vijay Lingam, Sujay Sanghavi, Sayan Ghosh, Linbo Liu, Jun Huan, Anoop Deoras, Gaurav Gupta, and Behrooz Omidvar-Tehrani. Enhancing language model agents using diversity of thoughts. *The Thirteenth International Conference on Learning Representations*, 2025.

[24] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[25] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

[26] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024.

[27] Victor May, Diganta Misra, Yanqi Luo, Anjali Sridhar, Justine Gehring, and Silvio Soares Ribeiro Junior. Freshbrew: A benchmark for evaluating AI agents on java code migration. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, 2025.

[28] Stoyan Nikolov, Daniele Codecasa, Anna Sjovall, Maxim Tabachnyk, Satish Chandra, Siddharth Taneja, and Celal Ziftci. How is google using ai for internal code migrations? *arXiv preprint arXiv:2501.06972*, 2025.

[29] Behrooz Omidvar Tehrani, Ishaani M, and Anmol Anubhai. Evaluating human-ai partnership for llm-based code migration. In *Extended abstracts of the CHI conference on human factors in computing systems*, pages 1–8, 2024.

[30] Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents, 2025.

[31] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 2023.

[32] Manish Shetty, Naman Jain, Adwait Godbole, Sanjit A. Seshia, and Koushik Sen. Syzygy: Dual code-test c to (safe) rust translation using llms and dynamic analysis, 2024.

[33] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

[34] Kaixin Wang, Tianlin Li, Xiaoyu Zhang, Chong Wang, Weisong Sun, Yang Liu, and Bin Shi. Software development life cycle perspective: A survey of benchmarks for code large language models and agents, 2025.

[35] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An Open Platform for AI Software Developers as Generalist Agents, 2024.

[36] Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. Repotransbench: A real-world benchmark for repository-level code translation, 2024.

[37] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

[38] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

[39] Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*, 2025.

[40] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[41] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

[42] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.
[43] Luke Yoffe, Alfonso Amayuelas, and William Yang Wang. Debunc: mitigating hallucinations in large language model agent communication with uncertainty estimations. *arXiv preprint arXiv:2407.06426*, 2024.
[44] Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, Dezhi Ran, Muhan Zeng, Bo Shen, Pan Bian, Guangtai Liang, Bei Guan, Pengjie Huang, Tao Xie, Yongji Wang, and Qianxiang Wang. Swe-bench-java: A github issue resolving benchmark for java, 2024.
[45] Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.
[46] Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. Scalable, validated code translation of entire projects using large language models, 2024.
[47] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*, 2024.
[48] Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, et al. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*, 2024.

A Data Collection

We show filter details for MigrationBench dataset collection, to augment Section 3.2.

A.1 F_4: Search for Base Commit ID H_b

When we go through the repository's commit history in the reverse order up to March 2024, we assume both `Java` versions and compiled class major versions are chronologically monotonic, which holds for most repositories. For initial hard coded java versions, while larger than `Java 8`, we fully leverage commit history for `pom.xml` files *only* to accelerate identifying the last commit with `Java 8`, instead of going through commit history for the whole repository. Whenever we encounter either hard coded `Java` versions or complied classes' major versions are less than the specification before finding a valid base commit, we terminate the search and exclude the repository from the benchmark dataset.

When it takes too long to run `mvn clean verify` or a repository is in a broken state with too many commits, and we're unable to find a valid base commit within 15 minutes, we also exclude it from MigrationBench. This is the best effort to keep the process to be more reproducible, take less compute time and preserve as many repositories as possible.

Due to the importance of base commit id H_b, for the *whole* MigrationBench dataset including both the `full` and `UTG` subsets, we re-run the process twice and only keep the ones reporting the same H_b across the two runs. Though we may miss repositories due to inconsistency introduced by e.g. `maven` throttling issues and runtime cutoff, it provides a *stronger* guarantee on both correctness and reproducibility.

B Benchmark Dataset Statistics: `full` and `selected` Comparison

In addition to dataset statistics shown in Section 3.4's Table 1, in Figures 3 and 4 we compare the discrete probability distribution function (PDF) and cumulative distribution function (CDF) respectively, for repositories in MigrationBench's `full` and `selected` subsets:

(1) Number of modules: It shows number of `pom.xml` files for each repository.
(2) LOC: It's a sum of total LOCs for all `*.java` files for each repository.
(3) Number of all or test-only `Java` files: It counts all or test-only (in the `src/test` directory) `*.java` files in a repository.
(4) Number of test cases: It parses number of test cases based on `mvn test -f .` command's output at the root directory, valid when non-negative, while a negative number means there are some issues running the tests or parsing the output, likely due to `maven` throttling issues though we try to mitigate it with multiple efforts.

C Evaluation

This section provides more details and extends Section 4.

C.1 Unit Test Coverage

Unit testing plays a pivotal role in ensuring functional equivalence during code migration, as it provides a robust mechanism to verify that the migrated code performs identically to its original counterpart, independent of whether they contain bugs or not. By isolating individual components of the code and testing them against predefined inputs and expected outputs, unit tests captures discrepancies introduced during migration, such as logic errors, incompatibilities or unintended side effects. This granular validation process builds confidence that the repository's new state in L_T retains the intended functionality as L_S and facilitates early detection and resolution of inconsistency issues.

Passing existing unit tests is already ensured in requirement r_1 by executing `mvn clean verify`. However, it is strongly recommended to generate additional unit tests to enhance test coverage, especially when the existing coverage is very low.

D Additional Experiments

We also perform additional experiments on a sampled subset of `migration-bench-full` to highlight that the `selected` subset is indeed a more challenging one. In this experiment, we sampled every 18 repositories from `migration-bench-full`, resulting in 283 repositories. We run hybrid approach with `Claude-4-5-Sonnet` on this sampled set.

Table 4: Maximal migration efficacy on a sampled subset of `migration-bench-full`. We use hybrid approach with `Claude-4-5-Sonnet`.

Dataset	η_{maximal} (%) ↑	Avg. # LLM Calls per Repo ↓
sampled full	72.79	42.30
`selected`	53.33	52.55

The experiments from Table 4 verified that the selected subset is more challenging in that it has lower success rate and that it requires more LLM calls to complete.

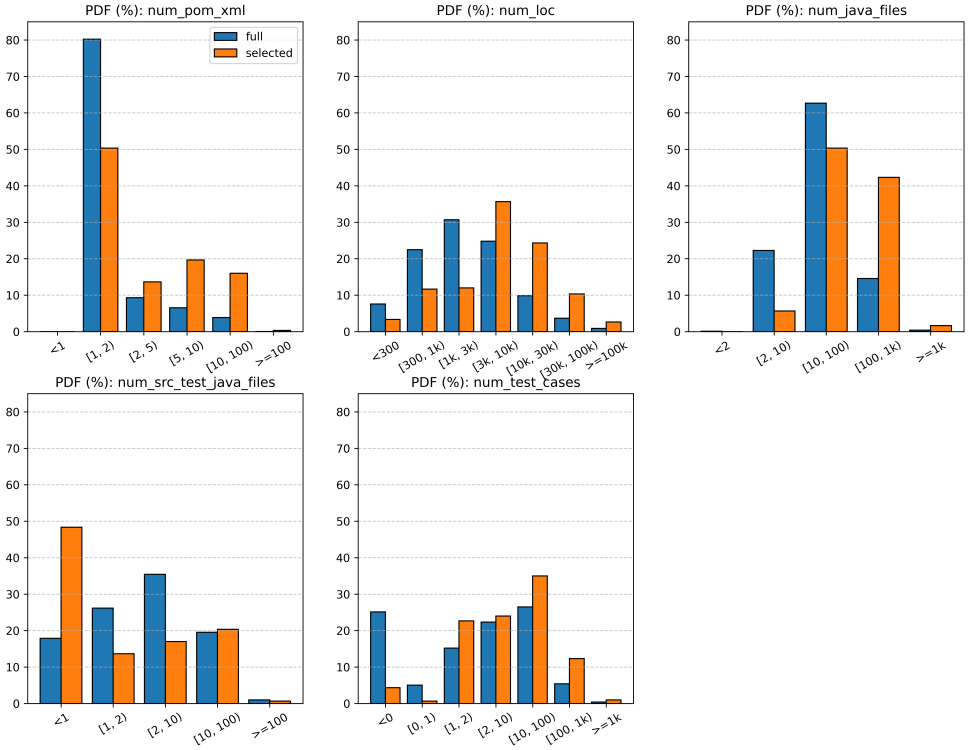


Figure 3: Discrete probability density function comparison for number of modules, LOC, all or test Java files and test cases for repositories in MigrationBench's `full` and `selected` subsets.

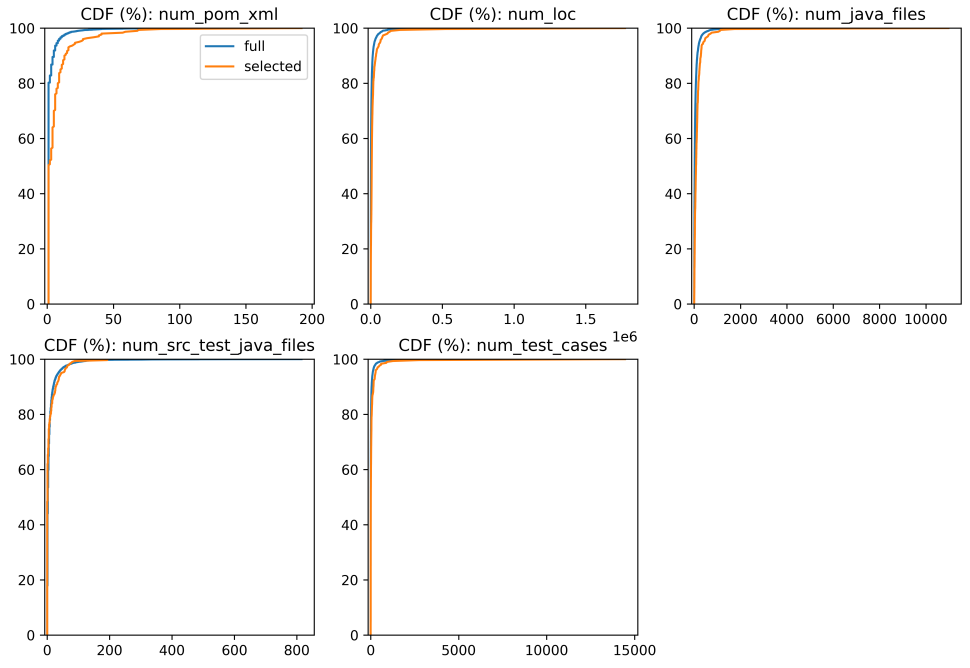


Figure 4: Cumulative distribution function comparison for number of modules, LOC, all or test-only Java files and test cases (dropping invalid negative values) for repositories in MigrationBench's `full` and `selected` subsets.