Skeleton-Guided-Translation: A Benchmarking Framework for Code Repository Translation with Fine-Grained Quality Evaluation

Xing Zhang[2*], **Jiaheng Wen**[3*], **Fangkai Yang**[1], **Yu Kang**[1†],
Pu Zhao[1], **Junhao Wang**[4], **Maoquan Wang**[1], **Yufan Huang**[1], **Shengyu Fu**[1], **Elsie Nallipogu**[1],
Qingwei Lin[1], **Yingnong Dang**[1], **Saravan Rajmohan**[1], **Dongmei Zhang**[1]
[1]Microsoft [2]Peking University [3]Zhejiang University [4]Tongji University

Abstract

Code translation benchmarks are essential for evaluating the accuracy and efficiency of LLM-based systems. Existing benchmarks mainly target individual functions, overlooking repository-level challenges like intermodule coherence and dependency management. Recent repository-level efforts exist, but suffer from poor maintainability and coarse evaluation granularity. We introduce Skeleton-Guided-Translation, a framework for benchmarking Java-to-C# translation at the repository level, featuring fine-grained quality evaluation. It follows a two-step process: first translating repository "skeletons", then refining the entire repository guided by these skeletons. Based on this, we present TRANSREPO-BENCH , the first test-driven benchmark of high-quality Java repositories paired with C# skeletons, unit tests, and build configurations. Our adaptive unit tests support multiple and incremental translations without manual tuning, enhancing automation and scalability. We also propose fine-grained metrics that evaluate translation quality per test case, overcoming limitations of binary metrics in distinguishing build failures. Evaluations using TRANSREPO-BENCH reveal issues like broken cross-file references, showing that our structured approach reduces dependency errors and preserves interface consistency.

1 Introduction

Large language models (LLMs) are reshaping software development, driving system modernization and legacy code migration. For example, migrating C to Rust improves safety (Matsakis and Klock, 2014), and frameworks like TensorFlow require synchronized multi-language updates. Evaluating LLMs in migration tasks is key to assessing reliability. Benchmarks provide quantitative insights for comparison and improvement, but existing ones

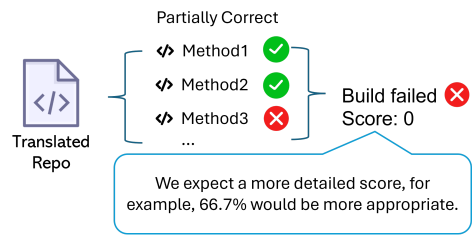

Figure 1: A more fine-grained quality evaluation to evaluate translated repositories is needed.

focus on function-level tasks or competition-style problems (Yan et al., 2023; Lu et al., 2021; Khan et al., 2024), ignoring real-world complexities. Repository-level translation is essential for managing dependencies, structure, and interconnected components (Jiao et al., 2023), requiring reliable benchmarks to assess model performance.

A major challenge in repository-level code translation is the absence of a systematic framework that enables fine-grained control over maintainability. For instance, updating parts of a Java-based SDK often requires re-translating large portions of the corresponding C++ codebase, making small changes costly. Without fine-grained control, maintainability suffers.

Another challenge is the lack of repository-level parallel corpora, complicating automated verification. Line-by-line metrics like codeBLEU (Ren et al., 2020) lack functional validation, and automatic test generation remains unreliable (Eniser et al., 2024). A practical alternative is translating unit tests from the source library for systematic validation. However, ensuring test accuracy and consistency with translated code interfaces is crucial for reliable verification.

The third challenge is that current metrics often miss nuanced translation outcomes, reducing usability. RepoTransBench (Wang et al., 2024), for example, uses a binary build success metric, ignoring partial successes. As Figure 1 shows, this over-

*These authors contributed equally to this work.
†Corresponding author.

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 18187–18198
November 4-9, 2025

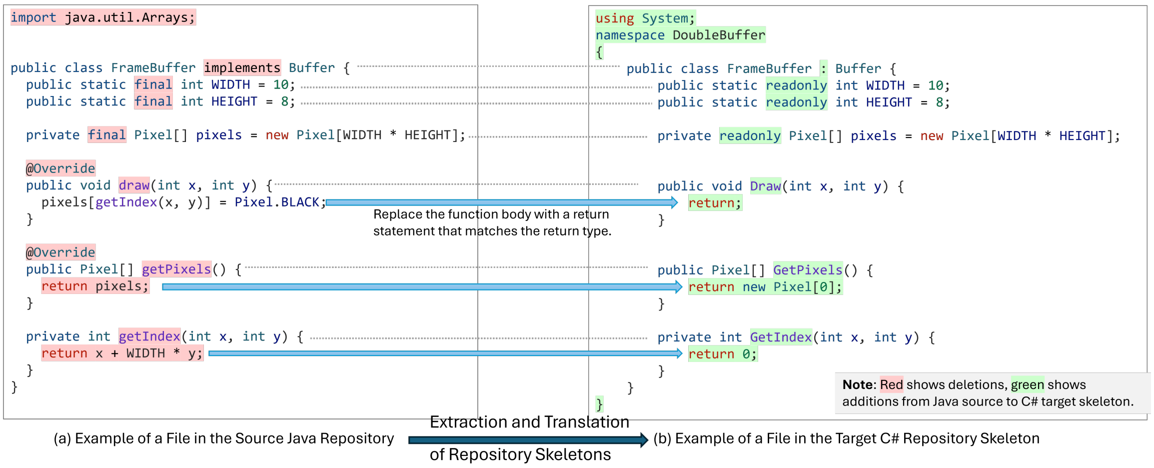

Figure 2: Example Code Snippets of Translation Input with Corresponding Skeleton

simplifies performance by neglecting cases where some components translate correctly while others fail. Schaeffer et al. (Schaeffer et al., 2023) warn that threshold-based metrics can create misleading performance leaps. In contrast, continuous metrics, such as the percentage of successfully translated modules (e.g., 66.7%), improve usability by identifying failures, guiding fixes, and providing smoother, more reliable insights.

Our Contributions

To address these challenges, we introduce Skeleton-Guided-Translation, a framework for benchmarking repository-level code translation with fine-grained quality evaluation. Our two-step process first translates the repository skeleton to define structure and interfaces, then populates it while indexing dependencies for unit tests. Skeletons are simplified versions of repositories with preserved structure and method signatures, but with method bodies replaced by defaults (e.g., return null). By providing unified unit tests, this design facilitates fair and consistent comparison across models during evaluation. Based on our framework, we present TRANSREPO-BENCH , the first test-driven benchmark to provide fine-grained evaluation, overcoming limitations of existing benchmarks. Specifically:

- We introduce Skeleton-Guided-Translation,[1] a novel framework for benchmarking repository-level code translation with fine-grained evaluation metrics. Complementing this, our benchmark TRANSREPO-BENCH , the first test-driven repo-level translation benchmark, provides a fine-grained evaluation by scoring individual test cases based on unit tests and their associated code, offering more meaningful feedback than binary metrics.
- High-Quality Open-Source Repository Benchmark: TRANSREPO-BENCH features high-quality open-source Java libraries and their C# translations, including unit tests and configurations. Designed for translation and fine-grained evaluation, it enables researchers to assess models in realistic repository-level scenarios.
- Evaluation of Advanced Models: TRANSREPO-BENCH is validated through extensive evaluations of classic and state-of-the-art models, offering detailed performance analysis. Our benchmark reveals that even SOTA LLMs reach only 26.65% accuracy under realistic repository conditions.

2 Motivation

In this section, we use an example to illustrate the challenges involved in building a repository-level code translation benchmark and explain our solutions more effectively.

2.1 Challenges in Repository Translation

Lack of a Systematic Translation Framework. Figure 2 presents an example of LLM-based Java-to-C# translation, underscoring the need for a systematic framework. Suppose the Java code in Figure 2(a) has already been translated. If a new method is later added to the Java FrameBuffer class, re-translating the updated code with an LLM is

[1]The source code implementing Skeleton-Guided-Translation, along with all code samples in our benchmark TRANSREPO-BENCH , are available at https://anonymous.4open.science/r/TransRepo-bench.

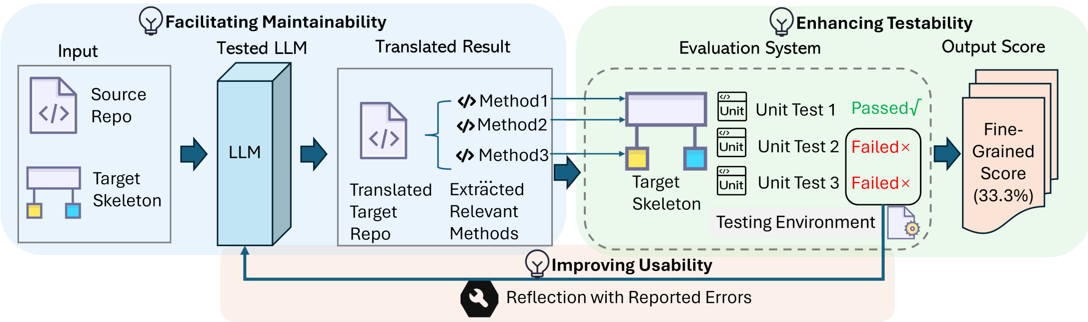

Figure 3: Framework of Our Evaluator.

likely to produce an inconsistent interface compared to the previous version. This can invalidate existing unit tests due to mismatched signatures or missing targets. The root cause is the lack of fine-grained incremental translation, where even minor changes may require re-generating the entire class or related components.

Lack of Parallel Corpora. Repository-level translation struggles with misaligned source and target files, complicating cross-language verification. For example, validating the C# code translated from Java code (Figure 2(a)) is challenging without existing ground truth. One solution is translating high-coverage Java tests into C#, but preserving intent, coverage, and reliability remains difficult. How to translate a set of unit tests once and use them to evaluate multiple independent translations by LLMs—or to compare translations generated by different models—is also a significant challenge.

Lack of a Fine-Grained Evaluation Metric. Relying on coarse metrics (e.g., whether a repository builds) limits developers' ability to diagnose translation issues. For instance, if `Draw` is mistranslated by calling `getIndex` instead of `GetIndex`, the compilation will fail, making it impossible to evaluate correctly translated functions like `GetPixels`. This binary pass/fail approach obscures partial successes and forces manual debugging. Granular metrics—such as module-level correctness or function fidelity—would help pinpoint errors, streamlining debugging and refinement.

2.2 Solution: Standardizing Code Repository Translation with Fine-Grained Evaluation

Figure 3 illustrates our solution. To align translation with testing and enable fine-grained evaluation, we introduce a target repository "skeleton" during translation. This guides LLMs to focus on accurate dependencies and interfaces. The skeleton is incrementally populated with partial results, allowing execution-based assessment of translation quality.

Facilitating Maintainability. Figure 2(b) illustrates a "target C# repository skeleton" in our framework. Unlike the fully translated Java code in Figure 2(a), this skeleton defines interfaces while leaving method bodies mostly empty. This approach improves maintainability: the C# skeleton enables incremental updates by aligning interfaces first, avoiding full re-translation.

Enhancing Testability. Building unit tests on these skeletons significantly improves testability. Because the structural and interface definitions in both repositories match, any unit tests originally designed for the Java code can be adapted to validate the C# skeleton. Provided the translated C# code is inserted into the appropriate skeleton methods, it can be reliably evaluated by the unit tests.

Improving Usability. The framework's fine-grained control improves usability by enabling targeted verification. If `Draw` is mistranslated and fails to compile, unit tests for `GetPixels` and `GetIndex` can still run within the skeleton (Figure 3). This ensures their correctness despite errors elsewhere. Unlike coarse build-or-fail metrics, skeleton-based testing reveals partial successes, streamlining debugging and evaluation.

3 TransRepo-bench Benchmark

As shown in Figure 3, users receive the source repository and target skeleton, guiding LLMs to generate a complete target repository. Correctness is verified using the target's unit tests within the testing environment. This section presents the benchmark content, details TransRepo-bench 's construction, and introduces our fine-grained evaluation design.

3.1 Benchmark Overview

Each TransRepo-bench translation task includes a source repository and its evaluation setup,

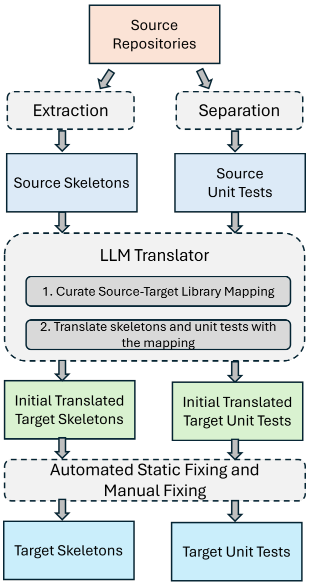

Figure 4: Benchmark construction workflow from extraction to final target skeletons and unit tests via mapping, translation, and fixing.

structured as <*source repository, target skeleton, target unit tests, testing environment*>. While we currently focus on Java-to-C# translation in our experiments, the proposed test-anchored skeleton methodology is language-agnostic and can generalize to other language pairs such as Python-Rust.

As shown in Figure 2, the translation task input includes Java source repositories for translation and a target repository skeleton, which serves as an interface "contract" for evaluation. This skeleton retains the original file structure, dependencies, and static values but replaces all functions with trivial implementations (e.g., a single return statement) to ensure successful compilation. The evaluation setup consists of unit tests for the target repository and the required testing configuration files.

TRANSREPO-BENCH includes 13 tasks for translating code repositories. Appendix A.1 provides details on repository features like class, method, and line counts, plus test coverage. The data highlights diverse complexities, from small repositories to large ones with extensive methods and coverage, ensuring robust evaluation.

3.2 Benchmark Construction

This section details the benchmark construction process (Fig. 4). We first describe source dataset collection (§3.2.1), then outline skeleton extraction and translation (§3.2.2). Next, we explain unit test acquisition (§3.2.3) and conclude with testing environment setup (§3.2.4). Overall, the construction process required approximately 340 person-hours of manual and semi-automated effort.

While users of TRANSREPO-BENCH do not need to construct skeletons manually, building new benchmarks based on our framework does involve generating new skeletons and tests. This process—described below—includes manual validation and test environment setup, which ensures high-quality evaluation infrastructure.

3.2.1 Source Repository Collection

The source dataset is curated from open-source GitHub projects meeting these criteria: (1) 100+ stars, (2) a testing workflow, and (3) locally passing tests. We chose a mature and well-tested collection of repositories from *java-design-patterns*, a Java library featuring comprehensive design pattern implementations and reliable test execution.

3.2.2 Skeleton Extraction and Translation

Repository skeletons are structural templates that preserve the complete architecture of a repository while replacing all method implementations with minimal, compilable stubs. Formally, for a repository R with classes $C = \{c_1, c_2, ..., c_n\}$, a skeleton $S(R)$ maintains: (1) all class declarations and inheritance hierarchies, (2) complete method signatures including parameters and return types, (3) field declarations and static members, (4) package/-namespace structures, and (5) import/using statements.Function bodies return type-matching placeholders (e.g., `return 0;` for `int`, `return null;` for `objects`). Constructors are left empty, and static blocks retain only assignments.

The skeleton construction follows a systematic process. First, we parse source files using Tree-sitter to generate abstract syntax trees (ASTs). Next, we traverse the ASTs to extract structural elements—classes, methods, fields, and their relationships. We then replace method bodies with type-appropriate default returns: primitive types return default values (0, false), reference types return null, void methods have empty bodies, and constructors preserve only essential field initializations. This process ensures compilable code while removing all business logic, enabling fair comparison across different LLM translations by providing identical structural baselines.

Skeletons are translated into the target language using GPT-4o, but most initially fail to compile, requiring manual fixes. As shown in Appendix A.1, "Skeleton Fix Time" quantifies this effort, averaging

105 minutes per repository. To ensure correctness, two experienced engineers manually verified the structural and functional equivalence between Java and C# skeletons across all 13 repositories over 72 person-hours. This validation focused solely on structural consistency—no manual code translation occurred, with all translation performed by LLMs.

Our framework assigns three roles to skeletons: (1) for benchmarking, we provide pre-built skeletons to ensure consistent comparisons across different LLMs, requiring no user effort; (2) for creating new benchmarks, users need to build new skeletons for target languages, either manually or semi-automatically using our provided tools; and (3) in real-world translation scenarios, skeletons are optional but significantly improve structure preservation, support incremental translation, and enable systematic unit testing.

3.2.3 Unit Test Translation

We translate source repository unit tests into the target language using GPT-4o and adapt them to the NUnit framework. However, most initially fail to compile due to API differences and naming conventions, requiring manual fixes to ensure correct validation. To verify semantic consistency, we executed Java tests on the Java skeleton and translated C# tests on the C# skeleton, confirming identical pass/fail results across both versions. The unit tests achieve 96.14% Java code coverage using JaCoCo and 94.8% C# line coverage with Coverlet, ensuring comprehensive validation of translated code.

3.2.4 Testing Environment Construction

We establish a testing environment by defining a Docker image with necessary dependencies and build tools. For each repository, we create a YAML build configuration file for the translated C# project, adapted from the original Java build file. This process involves mapping Maven/Gradle dependencies to NuGet packages and adjusting build scripts for the .NET ecosystem.

While GPT-4o assists in converting build configurations, manual refinement ensures functionality. To reduce manual effort and enhance framework usability, we provide supporting resources: static repair scripts for common skeleton and test issues, automated configuration generators for C# projects, and dependency mapping tables for common Java-to-C# library conversions. These tools significantly improve efficiency, though some manual intervention remains necessary for complex cases.

Model	Build Rate (%)			Unit Test Pass Rate (%)		
	Iteration1	Iteration2	Iteration3	Iteration1	Iteration2	Iteration3
GPT-4-turbo	60.54	66.31	50.00	15.59	18.16	11.25
GPT-4o	58.17	57.34	57.34	17.97	14.32	16.03
GPT-4o-mini	49.31	41.13	44.98	10.16	12.03	12.03
GPT-o1-mini	50.00	59.18	52.06	17.35	17.35	15.70
DeepSeek-v3	52.88	71.14	71.14	16.06	17.56	17.56
DeepSeek-r1	59.83	72.13	73.32	15.59	19.83	19.83
Claude-3.5	54.92	51.64	44.26	15.66	15.13	10.01
Qwen-plus	59.32	59.53	56.73	17.31	18.08	16.68

Table 1: Build rates (%) and Unit test pass rates (%) for different repositories across various models.

3.2.5 Framework Generalization

Although our implementation focuses on Java-to-C# translation due to their mature unit testing ecosystems (JUnit/NUnit), the methodology generalizes to any language pair meeting two criteria: (a) availability of test framework mappings, and (b) existence of behavioral validation mechanisms. The core language-agnostic components—skeleton guidance, file-level LLM translation, and unit test evaluation with dependency resolution—transfer directly to other language pairs.

3.3 Fine-Grained Evaluation Metrics Design

To refine user-translated code evaluation, we use unit tests for scoring. Prior attempts to translate entire repositories often failed at compilation, preventing test execution. Pan et al. (Pan et al., 2024) report 77.8% of large-model translation failures stem from compilation errors, obscuring correct translations and hindering evaluation. To mitigate this, we extract and execute test-relevant code within a guaranteed-compilable skeleton. Translated functions are inserted, then built and tested using dotnet build and dotnet test, ensuring granular scoring unaffected by unrelated errors.

Our evaluation uses two metrics: *build success rate*, the fraction of compilable unit tests, and *unit test success rate*, the fraction of passing tests among those that compile. We average these scores across libraries for an overall performance measure. The core challenge is extracting relevant source code for each test. We instrument Java source code at the function level to track invoked code, then map it structurally to the corresponding C# code, ensuring accurate test execution.

4 Evaluation

We first analyze LLM performance on our benchmark, then highlight our framework's effectiveness in using repository skeletons for translation and fine-grained evaluation.

Repo Name	Build Success Rate (%)								Unit Test Pass Rate (%)							
	GPT				DeepSeek		Others		GPT				DeepSeek		Others	
	o1-mini	4o-mini	4o	4-turbo	v3	r1	Claude	Qwen	o1-mini	4o-mini	4o	4-turbo	v3	r1	Claude	Qwen
promise	44.4	44.4	44.4	0.0	44.4	44.4	44.4	44.4	22.2	11.1	22.2	0.0	11.1	33.3	11.1	11.1
table-module	100.0	76.2	95.2	100.0	100.0	100.0	76.2	100.0	4.8	4.8	4.8	9.5	9.5	9.5	4.8	9.5
double-buffer	57.1	57.1	57.1	57.1	85.7	92.9	57.1	100.0	71.4	57.1	57.1	57.1	71.4	85.6	57.1	42.9
decorator	0.0	0.0	0.0	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	33.3	0.0	0.0	0.0	0.0
producer-consumer	0.0	0.0	0.0	100.0	100.0	100.0	100.0	0.0	0.0	0.0	0.0	33.3	33.3	33.3	33.3	0.0
double-dispatch	70.8	12.5	70.8	45.8	95.8	95.8	12.5	100.0	12.5	0.0	12.5	12.5	33.3	33.3	0.0	16.7
partial-response	100.0	100.0	100.0	100.0	60.0	70.0	100.0	60.0	20.0	0.0	20.0	20.0	0.0	20.0	20.0	0.0
converter	90.0	80.0	100.0	100.0	100.0	100.0	100.0	100.0	20.0	0.0	20.0	20.0	20.0	20.0	20.0	20.0
caching	80.0	100.0	100.0	50.0	50.0	50.0	50.0	90.0	40.0	0.0	10.0	0.0	10.0	10.0	0.0	40.0
unit-of-work	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	50.0	50.0	50.0	30.0	50.0	50.0	50.0	50.0
game-loop	77.8	88.9	77.8	100.0	88.9	100.0	100.0	77.8	33.3	33.3	55.6	11.1	33.3	33.3	33.3	33.3
type-object	88.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
bytecode	81.8	81.8	100.0	9.1	100.0	100.0	81.8	81.8	27.3	9.1	27.3	9.1	18.2	18.2	27.3	27.3
Average	**68.52**	**57.00**	**65.03**	**66.31**	**71.14**	**73.32**	**63.23**	**65.70**	**22.42**	**12.72**	**21.50**	**18.15**	**22.32**	**26.65**	**19.76**	**19.29**

Table 2: Build rates (%) and Unit test pass rates (%) for different repositories across grouped models.

4.1 Model Performance on TRANSREPO-BENCH

We evaluate the performance of state-of-the-art LLMs on the task of translating code repositories from Java to C#. Next, we conduct a failure analysis based on the experimental results.

4.1.1 Model Selection

We evaluated eight state-of-the-art LLMs for code repository translation: GPT-4o, GPT-4o-mini, GPT-4-turbo, Qwen-plus-1220, Claude-3.5-sonnet-20240620, DeepSeek-v3, DeepSeek-r1, and GPT-o1-mini. GPT-4o variants are efficient, general-purpose models. Qwen-plus-1220 and Claude-3.5-sonnet balance general and specialized reasoning. DeepSeek-v3 focuses on code understanding and transformation, while DeepSeek-r1 is a compact, efficient model with strong reasoning. GPT-o1-mini is lightweight and well-rounded, optimized for structured thinking.

4.1.2 LLMs Performance

Table 1 compares LLM performance over three iterations using Build Rate and Unit Test Pass Rate. DeepSeek-v3 improves consistently, achieving the highest Build Rate (71.14%) and a competitive Unit Test Pass Rate (17.56%) in Iteration 3. GPT-4-turbo starts strong (60.54%) but declines to 50.00%, with its Unit Test Pass Rate dropping to 11.25%. GPT-4o remains stable at 57.34% Build Rate, with minor fluctuations in Unit Test Pass Rate (16.03%). GPT-4o-mini and Claude-3.5 underperform, with declining Build Rates and inconsistent trends.

DeepSeek-r1 outperforms DeepSeek-v3, achieving the highest Build Rate (73.77%) and Unit Test Pass Rate (19.83%) in Iteration 3. GPT-o1-mini also improves, peaking at 59.18% Build Rate and maintaining a solid 15.7% Unit Test Pass Rate. Overall, DeepSeek-r1 is the most robust, followed by DeepSeek-v3, while other models struggle to maintain performance.

The results indicate that iterative refinement may not improve performance due to error accumulation, as early mistakes can amplify when models fail to distinguish helpful feedback from noise.

Build Rates. Table 2 shows DeepSeek-r1 (73.32%) and DeepSeek-v3 (71.14%) leading, followed by GPT-o1-mini (68.52%), GPT-4-turbo (66.31%), and Qwen-plus (65.70%). GPT-4o (65.03%) and Claude-3.5 (63.23%) perform slightly lower, with GPT-4o-mini (57.00%) trailing. DeepSeek-r1's strong performance suggests robust translation capabilities.

Unit Test Pass Rates. DeepSeek-r1 (26.65%) leads, followed by DeepSeek-v3 (22.32%), GPT-o1-mini (22.42%), and GPT-4o (21.50%). Claude-3.5 (19.76%) and Qwen-plus (19.29%) perform slightly lower, with GPT-4o-mini (12.72%) at the bottom. DeepSeek-r1 and GPT-o1-mini show stronger runtime behavior preservation.

4.1.3 Failure Analysis

Figure 5 shows error distribution and reduction over three iterations, demonstrating iterative refinement. The most frequent category, Runtime Errors, dropped from 439 in Iteration 1 to 428 in Iteration 3, reflecting ongoing improvements. Other common errors, including `CS0246` (missing type/namespace), `CS1061` (missing member), and `CS0103` (undefined variable/name), also declined, indicating effective correction. For instance, `CS0106` fell from 23 to 16, and `CS1061` from 23 to 17. The inconsistent decrease in `CS0103` and `CS0246` may result from newly introduced variables or dependencies

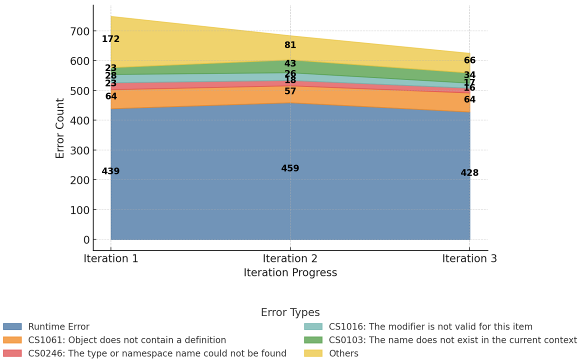

Figure 5: Changes in Error Proportions

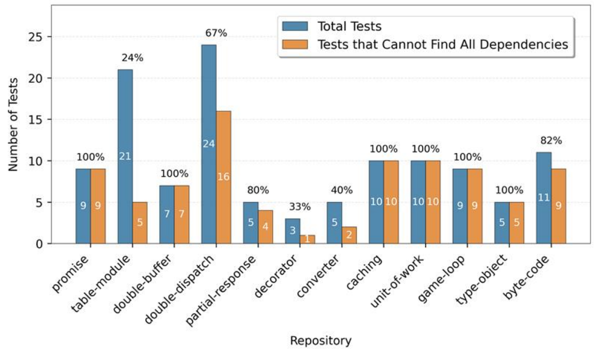

Figure 6: Missing Dependencies in Unit Tests Due to the Absence of Skeletons

Repo	**Build Score (%)**		**Unit Test Score (%)**	
	RepoTransBench	Ours	RepoTransBench	Ours
bytecode	100	44.4	81	22.2
caching	0	95.2	0	4.8
converter	0	57.1	0	57.1
decorator	0	0.0	0	0.0
double-buffer	0	0.0	0	0.0
double-dispatch	0	70.8	0	12.5
game-loop	0	100.0	0	20.0
partial-response	0	100.0	0	20.0
producer-consumer	0	100.0	0	10.0
promise	0	100.0	0	50.0
table-module	0	77.8	0	55.6
type-object	0	0.0	0	0.0
unit-of-work	100	100.0	30	27.3

Table 3: Comparison of RepoTransBench and FineEval evaluation methods on each repository.

Iteration Time	**Build Rate (%)**		**Unit Test Pass Rate (%)**	
	With Skeletons	Without Skeletons	With Skeletons	Without Skeletons
Iteration1	58.17	3.3	17.97	3.3
Iteration2	57.34	3.3	14.32	3.3
Iteration3	57.34	3.3	16.03	3.3

Table 4: Comparison of Build Rate and Unit Test Pass Rate of GPT-4o with and without Skeleton

Iteration Time	**Build Rate (%)**		**Unit Test Success Rate (%)**	
	Coarse-Grained Feedback	Ours	Coarse-Grained Feedback	Ours
Iteration-1	39.34	58.17	9.09	17.97
Iteration-2	50.00	57.34	13.94	14.32
Iteration-3	45.45	57.34	13.16	16.03

Table 5: Comparative Experiment on Coarse-Grained vs. Our Fine-Grained Feedback for Usability Validation.

lacking definitions. The total error count fell from 747 to 619, showing improved resolution of syntactical and logical errors. Common failure patterns are detailed in Appendix A.2.

4.2 TRANSREPO-BENCH Effectiveness

This section aims to validate (1) the fineness of our evaluation mechanism, (2) the necessity of incorporating skeletons in the translation process, and (3) the fulfillment of the three mentioned requirements.

4.2.1 Validating Evaluation Fineness

Our evaluation provides a finer, more comprehensive assessment of repository translation. Unlike RepoTransBench (Wang et al., 2024), which evaluates entire projects without skeletons, our method scores components individually, preventing single errors from invalidating correct translations. As Table 3 shows, RepoTransBench scores 0 on most tasks, successfully evaluating only two of thirteen. In contrast, our approach assigns scores even when compilation fails, achieving 100% success for unit test-related segments. This fine-grained evaluation recognizes partial successes rather than dismissing them due to isolated errors.

4.2.2 Proving Skeleton Necessity

The second experiment confirms that providing target repository skeletons is essential for translation. Table 4 shows that omitting skeletons drastically lowers build success and unit test pass rates. This is due to unresolved inter-file dependencies and interfaces, which hinder identifying functions under test. As Figure 6 illustrates, missing skeletons cause many unresolved dependencies, dropping all build and test scores to zero. For some repositories, dependencies become completely unresolvable without skeletons, highlighting their crucial role in enabling accurate evaluation.

4.3 Validating Three Key Requirements for Repository-Level Translation

As proposed in Section 2.2, our Skeleton-Guided-Translation meets three requirements. Testability is validated through large model evaluation, so we focus on maintainability and usability.

Maintainability. Our maintainability experiment evaluates how Skeleton-Guided Translation helps LLMs perform incremental Java-to-C# translation, improving repo-level maintainability. It translates

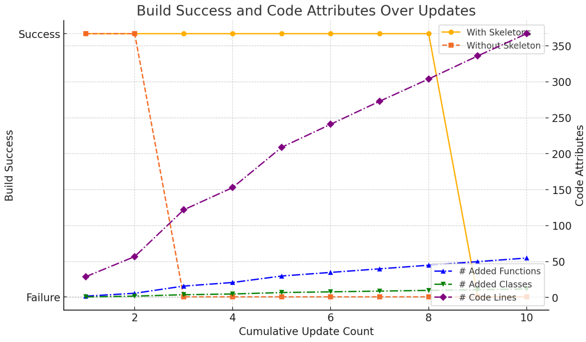

Figure 7: Build Success Rates for Incremental Translation with/without Skeleton

only necessary updates, avoiding redundant C# changes. We assessed the bytecode repository by measuring cumulative build success rates over ten incremental tasks across five trials. The first approach updated the skeleton before translation; the second translated directly without skeleton guidance. Figure 7 shows that the skeleton-guided method maintains successful builds even after eight updates and 45 new functions, while the unguided method fails around the third update. This demonstrates the effectiveness of skeletons in supporting incremental translation.

Usability. Table 5 compares coarse- and fine-grained feedback for improving translated libraries. Coarse feedback relies on holistic build and test evaluations, while fine-grained feedback provides targeted error insights. Results show that fine-grained feedback consistently improves build rates and unit test success, validating its effectiveness in model-guided code refinement.

Summary. These experiments collectively establish that our method is superior in two key aspects:

- Our evaluation mechanism is more granular and comprehensive, capturing the quality of translation even when partial failures occur.
- Skeletons are crucial for dependency resolution and accurate evaluation.
- Our Skeleton-Guided-Translation meets three key requirements for repository-level code translation: maintainability, testability, and usability.

5 Related Work

5.1 Code Translation

Code translation preserves semantics while converting languages. Rule-based compilers (e.g., Babel, Roslyn) handle simple cases but fail on complex constructs. AI-driven methods go further. Many studies (Tang et al., 2023; Roziere et al., 2020; Rozière et al., 2022; Yin et al., 2024; Yang et al., 2024; Jiao et al., 2023; Jana et al., 2024; Di et al., 2024; Tipirneni et al., 2024; Yan et al., 2023) focus on short code from competitive programming (Puri et al., 2021; Lu et al., 2021), educational platforms (Yan et al., 2023; Ahmad et al., 2023), or custom tasks (Liu et al., 2023; Chen et al., 2021). Some (Pan et al., 2024; Eniser et al., 2024; Zhang et al., 2023) tackle longer code (100+ lines) but with limited success. Novel training strategies (Roziere et al., 2020; Rozière et al., 2022; Szafraniec et al., 2023; Jana et al., 2024; Tipirneni et al., 2024) may enhance our approach, alongside prompting (Tang et al., 2023) and repair methods (Yin et al., 2024). Adapting automated program repair (Xia et al., 2023; Kong et al., 2024) could help with translation-specific I/O errors. SYZYGY (Shetty et al., 2025) translates C to safe Rust using LLM-driven code generation and dynamic analysis. Bhattarai et al. (Bhattarai et al., 2024) proposed a few-shot retrieval-based translation method, while Tao et al. (Tao et al., 2024) used an intermediary language (Go) to aid translation.

AlphaTrans (Ibrahimzada et al., 2024) is a neuro-symbolic framework for repository-level code translation using program analysis and dynamic testing. Shiraishi et al. (Shiraishi and Shinagawa, 2024) improved C-to-Rust translation with context-aware segmentation, and Oxidizer (Zhang et al., 2024) ensures functionality through feature mapping and unit tests. However, AlphaTrans struggles with semantic alignment in test translation and rigid syntax rules. Our method solves these by validating unit tests on both source and target skeletons and using LLMs to translate skeletons directly.

5.2 Code Translation Benchmarks

Benchmarks are crucial for evaluating code translation. Early ones used small, manually curated function pairs, while modern benchmarks cover large datasets across diverse languages. AdvBench (Robey et al., 2021) evaluates TransCoder on Java, C++, and Python using BLEU, Exact Match (EM), and Execution Accuracy. CodeNet (Puri et al., 2021) provides 14 million samples in 50 languages for training and evaluation. Task-specific benchmarks like CodeXGLUE (Lu et al., 2021) ensure functional correctness but often miss niche languages and system-level complexities. RustRepoTrans (Ou et al., 2024) first includes repository-level Rust dependencies, revealing a 41.5%-56.2%

performance drop, highlighting real-world challenges in dependency and cross-file handling.

RepoTransBench (Wang et al., 2024) benchmarks repository-level translation with 100 repositories and automated tests, addressing configuration, resource handling, and test migration. However, our approach overcomes its limitations: (1) No Skeleton Framework – Lacking skeletons, it struggles with interface constraints, leading to misalignments. Our skeleton-based method ensures better control and adaptability. (2) No Test Verification – It lacks robust test result checking, while we validate unit tests on both source and target skeletons for reliable evaluation. (3) Coarse-Grained Evaluation – It executes tests without isolating dependencies, compounding errors. Our approach isolates dependencies, enabling finer-grained assessment and reducing error propagation.

6 Conclusions

We present Skeleton-Guided-Translation and the TRANSREPO-BENCH benchmark to address the challenges of evaluating LLM-based repository-level code translation. We provide the "skeletons" that preserve file structures and interfaces, enabling fair and consistent evaluation of different LLMs with unified unit tests. Moreover, we offer fine-grained evaluation through detailed error localization, moving beyond simple pass/fail outcomes.

Our evaluation of TRANSREPO-BENCH, a set of Java repositories covered by high-quality tests, shows that even SOTA LLMs reach only 26.65% accuracy. Using skeletons prevents partial errors from affecting correct modules, providing a fine-grained assessment of build success and test pass rates. We also demonstrate the benefits of using skeletons to maintain interface consistency, enable fine-grained quality assessment, and support incremental translation.

7 Limitations

This study primarily focuses on evaluating repository-level translations between Java and C# using Skeleton-Guided-Translation, and does not confirm its generalizability to other language pairs (e.g., C++, Python, Rust). . Moreover, the experimental data is drawn from open-source projects with relatively high test coverage. While this offers some insight into how the approach might function in real-world scenarios, performance may degrade in extremely large or complex codebases with highly customized dependencies. Additionally, in order to maintain automation and control, we require the use of skeletons (and subsequent manual fixes) in the evaluation process, which may not fully capture more dynamic environments involving multi-user collaboration or frequent version updates. Lastly, certain unit tests still required manual patches before execution, somewhat limiting both efficiency and objectivity. Future research might explore automated repair techniques or adaptive testing configurations to further enhance evaluation reliability.

8 Ethical Considerations

The proposed method can significantly facilitate cross-language code migration and reuse but also introduces ethical and societal considerations. First, using large language models for automated code translation raises potential concerns about over-collection or misuse of proprietary software code, underscoring the need to address intellectual property and confidentiality agreements. Second, model-generated translations may contain hidden flaws or security vulnerabilities, and blindly deploying them into production risks exacerbating existing system vulnerabilities. Third, biases and metric selection in both the model training and evaluation processes may inadvertently cause disparities for certain languages or developer communities. To mitigate these issues, researchers and practitioners should collect data responsibly, rigorously review and test model outputs, and enforce thorough security and quality assessments before integrating translated code into production environments.

References

Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2023. Avatar: A parallel corpus for java-python program translation. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 2268–2281, Toronto, Canada. Association for Computational Linguistics.

Manish Bhattarai, Javier E. Santos, Shawn Jones, Ayan Biswas, Boian Alexandrov, and Daniel O'Malley. 2024. Enhancing code translation in language models with few-shot learning via retrieval-augmented generation. *Preprint*, arXiv:2407.19619.

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. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Peng Di, Jianguo Li, Hang Yu, Wei Jiang, Wenting Cai, Yang Cao, Chaoyu Chen, Dajun Chen, Hongwei Chen, Liang Chen, Gang Fan, Jie Gong, Zi Gong, Wen Hu, Tingting Guo, Zhichao Lei, Ting Li, Zheng Li, Ming Liang, Cong Liao, Bingchang Liu, Jiachen Liu, Zhiwei Liu, Shaojun Lu, Min Shen, Guangpei Wang, Huan Wang, Zhi Wang, Zhaogui Xu, Jiawei Yang, Qing Ye, Gehao Zhang, Yu Zhang, Zelin Zhao, Xunjin Zheng, Hailian Zhou, Lifu Zhu, and Xianying Zhu. 2024. Codefuse-13b: A pretrained multilingual code large language model. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '24, page 418–429. ACM.

Hasan Ferit Eniser, Valentin Wüstholz, and Maria Christakis. 2024. Automatically testing functional properties of code translation models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 21055–21062. AAAI Press.

Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Repository-level compositional code translation and validation. *Preprint*, arXiv:2410.24117.

Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. 2024. Cotran: An llm-based code translator using reinforcement learning with feedback from compiler and symbolic execution. In *Frontiers in Artificial Intelligence and Applications*, volume 392, pages 4011–4018. IOS Press.

Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. 2023. On the evaluation of neural code translation: Taxonomy and benchmark. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, pages 1529–1541. IEEE.

Mohammad Abdullah Matin Khan, M Saiful Bari, Do Long, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pages 6766–6805. Association for Computational Linguistics.

Jiaolong Kong, Mingfei Cheng, Xiaofei Xie, Shangqing Liu, Xiaoning Du, and Qi Guo. 2024. Contrastrepair: Enhancing conversation-based automated program repair via contrastive test case pairs. *Preprint*, arXiv:2403.01971.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, page 943. Curran Associates Inc.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1.

Nicholas D. Matsakis and Felix S. Klock. 2014. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14. Association for Computing Machinery.

Guangsheng Ou, Mingwei Liu, Yuxuan Chen, Xin Peng, and Zibin Zheng. 2024. Repository-level code translation benchmark targeting rust. *Preprint*, arXiv:2411.13990.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, page 1–13. ACM.

Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio

Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *Preprint*, arXiv:2009.10297.

Alexander Robey, Luiz F. O. Chamon, George J. Pappas, Hamed Hassani, and Alejandro Ribeiro. 2021. Adversarial robustness with semi-infinite constrained learning. *Advances in neural information processing systems*.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems*, volume 33. Curran Associates, Inc.

Baptiste Rozière, Jie Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging automated unit tests for unsupervised code translation. In *Proceedings of the 10th International Conference on Learning Representations*.

Rylan Schaeffer, Brando Miranda, and Sanmi Koyejo. 2023. Are emergent abilities of large language models a mirage? In *Advances in Neural Information Processing Systems*, volume 36. Curran Associates, Inc.

Manish Shetty, Naman Jain, Adwait Godbole, Sanjit A. Seshia, and Koushik Sen. 2025. Syzygy: Dual code-test C to (safe) Rust translation using LLMs and dynamic analysis. Preliminary version accepted at LLM4Code 2025. arXiv preprint arXiv:2412.14234.

Momoko Shiraishi and Takahiro Shinagawa. 2024. Context-aware code segmentation for c-to-rust translation using large language models. *Preprint*, arXiv:2409.10506.

Marc Szafraniec, Baptiste Roziere, Hugh Leather, François Charton, Patrick Labatut, and Gabriel Synnaeve. 2023. Code translation with compiler representations. In *International Conference on Learning Representations*. In-Person Oral Presentation, Top 25% Paper.

Zilu Tang, Mayank Agarwal, Alexander Shypula, Bailin Wang, Derry Wijaya, Jie Chen, and Yoon Kim. 2023. Explain-then-translate: an analysis on improving program translation with self-generated explanations. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. Association for Computational Linguistics.

Qingxiao Tao, Tingrui Yu, Xiaodong Gu, and Beijun Shen. 2024. Unraveling the potential of large language models in code translation: How Far Are We? In *31st Asia-Pacific Software Engineering Conference*, APSEC '24. To appear.

Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. 2024. Structcoder: Structure-aware transformer for code generation. *Preprint*, arXiv:2206.05239.

Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. 2024. Repotransbench: A real-world benchmark for repository-level code translation. *Preprint*, arXiv:2412.17744.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. CodeTransOcean: A comprehensive multilingual benchmark for code translation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. Association for Computational Linguistics.

Aidan Z. H. Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. 2024. Vert: Verified equivalent rust transpilation with large language models as few-shot learners. *Preprint*, arXiv:2404.18852.

Xin Yin, Chao Ni, Tien N. Nguyen, Shaohua Wang, and Xiaohu Yang. 2024. Rectifier: Code translation with corrector via llms. *Preprint*, arXiv:2407.07472.

Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. 2024. Scalable, validated code translation of entire projects using large language models. *Preprint*, arXiv:2412.08035.

Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. Multilingual code co-evolution using large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023. Association for Computing Machinery.

A Appendix

A.1 Detailed Information of TransRepo-bench

Repo Name	Classes	Methods	Lines	Unit Test Coverage	Skeleton Fix Time (min)
promise	6	36	789	93.70%	270
table-module	3	8	494	100.00%	70
double-buffer	3	16	489	98.30%	25
decorator	3	10	351	96.50%	60
producer-consumer	4	8	372	96.40%	30
double-dispatch	15	55	985	98.60%	90
partial-response	2	5	382	90.10%	130
converter	3	8	367	98.80%	100
caching	10	63	1605	93.30%	270
unit-of-work	4	16	460	98.30%	30
game-loop	7	18	730	94.90%	60
type-object	6	20	704	96.20%	120
bytecode	4	17	624	94.70%	150

Table 6: Resulting Benchmark

Table 6 summarizes the key characteristics of our benchmark repositories, highlighting their diversity, high test coverage, and moderate adaptation costs. The selected repositories cover a wide range of software design patterns, ensuring a comprehensive evaluation of translation performance. The number of classes, methods, and lines of code varies significantly across repositories, reflecting different levels of complexity and structural diversity.

Additionally, unit test coverage is consistently high across the benchmark, demonstrating the robustness of the evaluation setup and ensuring that translated code can be rigorously tested. The skeleton fix time, while necessary to adapt the repository skeletons for evaluation, remains moderate across all repositories, indicating a reasonable effort in preparing the benchmark without excessive overhead. Overall, this benchmark provides a well-balanced dataset, offering diverse software structures, strong test coverage, and a practical adaptation cost, making it suitable for assessing translation performance across different codebases.

A.2 Common Failure Patterns

We explore the most common failure patterns encountered during large model-based code translation, focusing on their underlying causes, how they manifest in practice, and the strategies needed to address them. By analyzing these recurring issues, we aim to provide actionable insights for improving the accuracy and reliability of cross-language code conversion processes.

Static Variable Misalignment. A common translation issue is inconsistent static variable naming. For example:

```
public void Stop(){
    status = GameStatus.Stopped;
}
```

The C# code raised error `CS0117` due to incorrect translation of the enum member `Stopped`, which should follow C#'s uppercase convention, e.g., `STOPPED`. This mismatch stems from Java's mixed-case style. To prevent such errors, translators should apply capitalization-aware mappings.

Unresolved Names and Contextual Misinterpretations. Translation errors often stem from missing imports of contextual elements, causing errors like `CS0103` ("The name does not exist in the current context"). For example:

```
private int RandomInt(int min, int max){
    return ThreadLocalRandom.Current.Next(min, max +
        1);
}
```

In this case, the C# compiler failed because `ThreadLocalRandom` is not recognized in C#. Instead, C# provides a `Random` class with similar functionality. Translators must correctly identify equivalent libraries and methods in the target language or include necessary imports automatically.

Undefined Methods. Errors such as `CS1061` occur when the translated code references methods or properties that are undefined in the target language. For instance:

```
_wizards[wizard].SetWisdom(amount);
```

This snippet assumes a `SetWisdom` method in the `Wizard` class, but the translator didn't verify it. Enhancing cross-referencing and generating warnings can help resolve such semantic gaps.

Namespace and Duplicate Definitions. Another common error (`CS0101`) occurs when namespaces contain duplicate definitions due to repetitive code generation. Consider the following Java snippet:

```
public class Candy
{
    public Candy(string flavor) { }
}
```

If the translator generates multiple constructors with identical signatures for this class in C#, the compiler will flag a conflict, as C# enforces unique member definitions within a namespace or class. The solution involves ensuring that constructors or methods with overlapping signatures are merged or disambiguated during translation.

Runtime Logical Failures. Even after fixing compilation errors, logical inconsistencies in the translation can still cause runtime issues. For example:

```
private void Register(Weapon weapon, string
    operation){
    if (!_context.TryGetValue(operation, out var
        weaponsToOperate))
    {
        weaponsToOperate = new List<Weapon>();
    }
    weaponsToOperate.Add(weapon);
    _context[operation] = weaponsToOperate;
}
```

A null reference error occurs because the `_context` dictionary was uninitialized. Such runtime errors are hard to catch via static analysis, underscoring the need for robust runtime testing to detect logical flaws in translated code.