**Neuro-Symbolic Code Translation and Validation**

by

Ali Reza Ibrahimzada

Dissertation

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2026

Urbana, Illinois

Doctoral Committee:

Assistant Professor Reyhaneh Jabbarvand, Chair
Professor Darko Marinov
Associate Professor Sasa Misailovic
Dr. Saurabh Sinha, IBM Research
Professor Daniel Kroening, University of Oxford and Amazon

**Abstract**

Software systems often outlive the programming languages (PLs) in which they were first written. As hardware platforms, deployment environments, security expectations, and developer ecosystems evolve, organizations must repeatedly migrate valuable code from legacy or ill-suited languages to modern ones. Automated code translation promises to reduce this migration effort, but reliable repository-level translation remains difficult. Traditional transpilers encode useful language knowledge in their design, yet they require substantial language-specific engineering and often produce brittle, non-idiomatic code. Large language models (LLMs) produce more natural and idiomatic translations, yet they struggle on real repositories, where correctness depends on cross-file dependencies, library APIs, type systems, and developer-written tests.

This dissertation presents work on neuro-symbolic and agentic code translation and validation. To make repository-level translation more practical, this dissertation develops techniques that combine the generative abilities of LLMs with program analysis, execution feedback, test-based validation, and multi-agent workflows for translation and validation. These techniques expose the structure of real codebases, guide translation decisions with semantic context, and use validation not merely as a final check but as a source of feedback for detecting and repairing translation failures.

This dissertation presents contributions along two key directions. The first direction studies why LLM-based code translation fails in realistic settings. Although LLMs can often translate small, self-contained examples, real repositories require global reasoning about language-specific features, call chains, libraries, and tests. This dissertation presents an empirical study of these failures and develops a taxonomy of translation bugs. The taxonomy establishes a foundational understanding of LLM translation limitations and serves as a guide for designing practical tools to mitigate the most common failures.

The second direction develops techniques for translating, validating, and repairing repository-level code. It first presents AlphaTrans, a neuro-symbolic pipeline that decomposes a repository into smaller fragments, translates them in dependency-aware order, and validates partial translations in isolation while preserving the behavior exercised by existing tests. It then presents MatchFixAgent, a language-agnostic validation and repair framework that combines approximate semantic analyses with specialized agents for test generation and repair. Finally, it presents ReCodeAgent, an end-to-end multi-agent workflow that assigns analysis, planning, translation, and validation to dedicated agent scaffolds; ablation studies show that this division of labor is necessary, as monolithic single-agent instantiations cannot achieve comparable effectiveness.

Together, these contributions show that LLMs alone are not sufficient for dependable code translation, but can become effective components of larger systems when guided by program analysis and subjected to rigorous validation. The techniques in this dissertation move automated code translation from simple examples toward realistic software modernization, helping developers understand translation failures, produce more reliable translations, and reason more systematically about the correctness of translated code.

## Acknowledgements

I would like to express my deepest gratitude to my PhD advisor, Professor Reyhaneh Jabbarvand, for her invaluable expertise, patience, and unwavering support throughout my graduate studies. Her insightful guidance was instrumental in shaping my research journey and fostering my growth as a scholar. I am particularly grateful for the opportunity she provided in the summer of 2021 through the University of Illinois Urbana-Champaign's research internship program, which introduced me to software engineering research despite my lack of prior background. Her persistent belief in my potential has been a cornerstone of my journey, and I am honored to have developed as a researcher under her guidance.

I would also like to extend my sincere gratitude to my doctoral committee members for their invaluable time, thoughtful feedback, and steadfast commitment, all of which significantly strengthened this dissertation. I am especially grateful to Professor Darko Marinov, whose generous support and stimulating discussions deepened my understanding of my research, and to Dr. Saurabh Sinha, my mentor at IBM Research, whose guidance shaped me into a more capable and confident researcher. I also thank Professor Daniel Kroening for his generous support and advocacy during my job search at Amazon.

Over the years at Urbana-Champaign, I was incredibly fortunate to find people who turned a new city into a home. I am especially grateful to Bengü Süeda Şengül, one of the smartest, kindest, caring, and supportive people I've ever met. Her unwavering support and kindness carried me through some of the most uncertain and difficult moments of my PhD. I also cherish the friendships of Chirag Shetty (who I sincerely hope keeps his ACL intact for the rest of his life), Vimarsh Sathia, Henry Zhu, Ahan Gupta, Kevin Ros, Louis Schatzki (thank you for the World Cup ticket), Marc Canby, Kayvon Mazooji, Changshu Liu, Zijie Zhao, Yuxiang Wei, Yifeng Ding, Chenyuan Yang, Jiawei Liu, Yicheng Ouyang, Jun Yang, and Dylan Zhang. The journey through a PhD can be isolating and grueling, but because of all of you, mine was filled with laughter, warmth, and belonging.

I would also like to express my heartfelt gratitude to Professor Ali Çakmak, who first opened the door to research for me in the summer of 2019. His support and belief in me planted a seed that would eventually grow into this dissertation, and I am especially grateful that he took the time to forward UIUC's email about the undergraduate research internship in 2021—a small act that quietly changed the entire trajectory of my life.

My PhD internships brought me not only invaluable research experience, but also wonderful friends and collaborators whom I will carry with me long after this journey. At IBM Research, I had the privilege of working alongside Raju Pavuluri, Saurabh Sinha, John Rofrano, Rangeet Pan, Rahul Krishna (I still have your Logitech mouse), Boris Sobolev, Lambert Pougeum Wassi, and Michele Merler; their camaraderie and intellectual generosity made my time there deeply rewarding. At Amazon Web Services, I am grateful to Brandon Paulsen, Joey Dodds, Luan Pham, Zhenning Yang, Hui Guan, and Victor Nicolet, whose mentorship and warmth made me feel welcomed and inspired at every step.

I gratefully acknowledge the institutions whose financial support made this research possible. This work was supported in part by the National Science Foundation (NSF) grant CCF 22-38045 and the IBM-Illinois Discovery Accelerator Institute. I also thank Professor Elsa Gunter for her generous support during my final semester, at a particularly uncertain and stressful time.

Above all, I owe an immeasurable debt of gratitude to my parents—my mom, Freshta, and my dad, Abdul Hakim—whose unconditional love has been my greatest strength throughout this long and demanding journey. They never had the opportunity to walk the halls of a university themselves, yet they poured every ounce of their hope, sacrifice, and belief into my education. In their eyes, I always found the courage to keep going, especially in the moments when the path felt uncertain. This dissertation is, in many ways, a testament to their dreams as much as my own. I would also like to thank my sister Yalda and my brother Ferdows, whose unwavering encouragement and love have meant more to me than words can express. I carry all of you with me in everything I do.

*To my Mamani and Babayi, for everything*

## Contents

Chapter 1 Introduction . . . . . 11
1.1 Contributions . . . . . 14
1.2 Enduring Takeaways . . . . . 17
1.3 Dissertation Organization . . . . . 18
Chapter 2 Background and Related Work . . . . . 21
2.1 Related Work . . . . . 21
2.2 Research Gap . . . . . 27
Chapter 3 Research Problem . . . . . 34
3.1 Problem Statement . . . . . 36
3.2 Research Hypothesis . . . . . 39
3.3 Thesis Statement . . . . . 42
Chapter 4 Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code . . . . . 44
4.1 Introduction . . . . . 44
4.2 Empirical Setup . . . . . 47
4.3 LLM-Based Code Translation . . . . . 50
4.4 LLM-Based Translation Bugs . . . . . 53
4.5 LLM- vs. Non-LLM-based Translation . . . . . 66
4.6 Mitigating Translation Bugs . . . . . 69
4.7 Discussion . . . . . 74
4.8 Threats to Validity . . . . . 75
4.9 Related Work . . . . . 77
4.10 Concluding Remarks . . . . . 77
Chapter 5 AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation . . . . . 79
5.1 Introduction . . . . . 79
5.2 Challenges in Repository-Level Code Translation . . . . . 82
5.3 Overview of Approach . . . . . 84
5.4 Program Transformation and Decomposition . . . . . 86
5.5 Type Translation and Skeleton Construction . . . . . 89
5.6 Compositional Translation and Validation . . . . . 91

5.7 Empirical Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
5.8 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
5.9 Threats to Validity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
5.10 Concluding Remarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
Chapter 6 MatchFixAgent: Language-Agnostic Autonomous Repository-Level Code Translation Validation and Repair . . . . . . . . . . . . . . . . . . 116
6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
6.2 Limitations of Prior Work . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
6.3 MatchFixAgent . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
6.4 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
6.5 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 151
6.6 Threats to Validity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153
6.7 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153
Chapter 7 ReCodeAgent: A Multi-Agent Workflow for Language-agnostic Translation and Validation of Large-scale Repositories . . . . . . . . . . . . 155
7.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 155
7.2 Problem Definition and Architecture Design . . . . . . . . . . . . . . . . . 159
7.3 ReCodeAgent . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 161
7.4 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 170
7.5 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 183
7.6 Threats to Validity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 185
7.7 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 185
Chapter 8 Conclusion and Future Work . . . . . . . . . . . . . . . . . . . . . . 186
8.1 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 187
8.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 191
References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 196

# Chapter 1 Introduction

The lifespan of software often exceeds that of the programming language (PL) it was originally written in. For instance, critical enterprise systems, some decades old and written in languages like COBOL, Fortran, or hand-tuned C, must be regularly migrated to function on contemporary hardware and infrastructure [1], [2], [3], [4], [5]. Similarly, proprietary or domain-specific PLs, once suitable for a niche business problem, can become liabilities. This occurs when the developers who understand them retire, their support ecosystems lose security updates, or the organization shifts to a cloud-native model [6], [7], [8], [9], [10], [11], [12], [13]. Furthermore, even code written in a widely-used, modern PL can benefit from translation into a different PL that offers better performance, safety features, or a more suitable library ecosystem for the current application [14], [15], [16], [17].

The process of automatically converting a program from a *source* PL to a semantically equivalent program in a *target* PL, known variously as *source-to-source compilation*, *transpilation*, or *code translation*, has therefore been a persistent goal of programming languages and software engineering research for more than two decades. When it works, code translation can slash months of manual migration effort into hours, reduce human-introduced bugs, and unlock modern development practices on aging codebases. When it fails—as it does almost always, in subtle ways that evade immediate detection—the translated code silently misbehaves in production and the engineering team is left to debug a codebase that nobody has yet read.

Prior to the rise of large language models (LLMs), two families of techniques dominated code translation. The first, *rule-based transpilers*, relies entirely on program analysis rules written by hand. Tools such as C2Rust [18], CxGo [19], Sharpen [20], and Java2CSharp [21] fall into this category. Rule-based transpilers excel at preserving the syntactic and semantic structure of the source program, but they are fundamentally *brittle*: they target a single source–target pair, they cannot accommodate novel language features without additional manual engineering, they produce code that is frequently unsafe (e.g., C2Rust often emits Rust that is littered with `unsafe` blocks [22]), and their output is widely reported to lack idiomaticity and readability [22], [23]. The second family, *statistical and neural machine translation for code*, reformulated translation as a sequence-to-sequence problem [24], [25], [26], [27], [28], [29], [30], [31]. These models produced idiomatic output, but they were trained on small, curated corpora of aligned programs, and they struggled with translations that required any non-local reasoning about the source—e.g., identifier resolution across files, inheritance hierarchies, or library semantics.

Both families share a common limitation: they treat translation as a purely local transformation, and they do not scale to *real-world repositories*, with their tens of thousands of lines of code, their cross-file dependencies, and their reliance on third-party libraries whose APIs have no direct counterpart in the target PL.

The rise of LLMs—first general-purpose models such as GPT [32] and Claude [33], and then code-specialized models such as StarCoder [34], CodeGen [35], CodeGeeX [36], and DeepSeek-Coder [37]—fundamentally changed the outlook for automated code translation. For the first time, a single model could translate between PLs it had never been explicitly paired on during training, handle libraries whose APIs were only implicitly present in the training data, and produce code in the target PL that was often indistinguishable, at the line level, from hand-written code. Early results on crafted benchmarks were striking: GPT-4 in particular could translate between Python, Java, C++, and Go with success rates in the double digits or higher on crafted benchmarks [22].

These early results, however, papered over a deeper problem. Our own large-scale empirical investigation, reported in Chapter 4, shows that LLM-based translation of *real-world* projects is dramatically worse than LLM-based translation of *crafted benchmarks*. When GPT-4, the strongest model in our study, was tasked with translating two open-source command-line projects end-to-end (Apache Commons CLI and Python Click), only 8.1% of the translations passed the existing test suite; all other studied LLMs scored 0%. The reasons for this collapse are now well-understood: LLMs hallucinate class, method, and library names when the true names are not in the prompt [22]; they exhibit short, non-uniform attention over long contexts [38] and therefore cannot reason about inter-procedural dependencies even when the relevant code is included verbatim in the prompt; and they handle PL-specific features (constructor overloading, annotations, unsafe pointers, exception hierarchies) in idiosyncratic and inconsistent ways.

In short, LLMs on their own are necessary but *not* sufficient for real-world code translation. Some of the state-of-the-art from the program analysis world—symbol resolution, call graphs, type inference, dependency analysis—must be brought back in, in a carefully engineered composition that plays to each side's strengths.

This dissertation argues, and empirically demonstrates, that the right composition of LLMs and program analysis, when structured as a neuro-symbolic and ultimately agentic pipeline, can translate and validate real-world repositories across multiple programming language pairs. The argument unfolds as a four-thrust research arc that traces the path from *understanding* what goes wrong when LLMs translate code (Chapter 4), to *solving* repository-level translation for a single language pair with heavy symbolic assistance (Chapter 5), to *generalizing* the validation and repair halves of the pipeline across arbitrary language pairs (Chapter 6), and finally to *generalizing the entire pipeline* across language pairs via a principled multi-agent architecture in which analysis, planning, translation, and validation each receive a dedicated agent scaffold (Chapter 7)—a decomposition shown to be necessary because monolithic single-agent scaffolds cannot match its effectiveness. Each thrust closes one of the four research questions identified in Chapter 2 and establishes one of the four research hypotheses formalized in Chapter 3. The dissertation's contributions with their headline empirical results are summarized in §1.1, its enduring methodological takeaways in §1.2, and a chapter-by-chapter reading guide is given in §1.3.

This dissertation is written to serve three overlapping audiences. *Software engineering researchers* interested in automated program transformation, repository-level reasoning, and multi-agent software systems will find the novel algorithms, empirical methodology, and publicly released artifacts directly useful. *Practicing engineers* focused on migrating or modernizing production codebases will find quantitative evidence of what LLM-based translation can (and cannot) do today, a bug taxonomy that can double as a debugging checklist, and open-source pipelines (AlphaTrans, MatchFixAgent, ReCodeAgent) they can apply to their own projects. *Applied ML researchers* building code agents will find a careful study of the design space of multi-agent translation and validation workflows, including ablations that isolate the contribution of each agent and each tool.

### 1.1 Contributions

This dissertation makes five primary contributions, four of which correspond to the four research chapters (Chapter 4–Chapter 7) and the fifth of which consists of the open-source artifacts, benchmarks, and datasets released alongside this work.

**C1. The first large-scale empirical study and bug taxonomy for LLM-based code translation.** Chapter 4 reports 43,379 translations, produced by seven general and code LLMs (GPT-4, Llama 2, TheBloke-Vicuna, TheBloke-Airoboros, StarCoder, CodeGeeX, and CodeGen) over five PLs (C, C++, Go, Java, Python). Through 630 person-hours of manual labeling by eight labelers, we derive a 15-category, 5-group taxonomy of LLM translation bugs that generalizes across models, PLs, and benchmarks. We quantify the effectiveness of iterative prompt engineering as a mitigation strategy, showing an average improvement of only 5.5% across the studied LLMs (and 12% for GPT-4, the strongest). This is the first systematic analysis of real-world failures in LLM-based code translation, and the taxonomy is used as a debugging lens by every subsequent chapter.

**C2.** **AlphaTrans, the first technique for compositional translation and validation of repository-level code.** Chapter 5 introduces AlphaTrans, a neuro-symbolic compositional approach that unifies translation and validation and translates and validates entire real-world repositories—including their tests—between Java and Python. AlphaTrans is, to our knowledge, the first technique to translate a complete real-world repository, and the first to leverage language interoperability (via GraalVM) for *in-isolation* validation of translated fragments while the rest of the project remains in the source PL. On ten real-world Java projects, AlphaTrans achieves 96.40% syntactic correctness over 17,874 fragments, 27.03% runtime validation and 25.14% functional equivalence over 4,654 executable method fragments (rising to 99.2% syntactic correctness and 27.95% functional equivalence with GPT-4o, at an average cost of $14.39 per project). A human subject study shows that partial AlphaTrans translations can be brought to passing test suites in 20.1 hours on average, providing the first pragmatic dataset for downstream research on repository-level translation repair.

**C3.** **MatchFixAgent, the first language-agnostic agentic technique for translation validation and repair at the repository-level.** Chapter 6 introduces MatchFixAgent, a multi-agent, PL-agnostic framework that validates and repairs repository-level translations by combining lightweight, approximate semantic analyses (control-flow, data-flow, library API, exception, and specification analyses) with a test generator and repair agent and a separate verdict agent. Evaluated on 2,219 source–translation function pairs drawn from 24 real-world projects totaling over 900,000 lines of code across six PL pairs, MatchFixAgent produces a verdict on 99.2% of pairs (versus 71.6% for prior PL-specific techniques), repairs 50.6% of inequivalent translations (32.1% above the strongest prior baseline), and—the keystone of the contribution—requires only roughly 280 lines of PL-specific code to support a new PL pair, compared with several thousand lines for prior work. An ablation study demonstrates that removing either the semantic analyzer or the in-the-loop test generator component reduces verdict accuracy by up to 42.3% while *increasing* token usage, establishing both components as necessary.

**C4. ReCodeAgent, the first multi-agent, language-agnostic framework for repository-level translation and validation.** Chapter 7 introduces ReCodeAgent, a four agent framework—*Analyzer*, *Planning*, *Translator*, and *Validator*—that grounds every agent in PL-agnostic tooling (Tree-sitter and Language Server Protocols). Evaluated on 4,583 translation units drawn from 118 real-world projects totaling over 230,000 lines of code across four PL pairs (C–Rust, Go–Rust, Java–Python, Python–JavaScript), ReCodeAgent achieves 99.4% compilation and 86.5% test pass rates (2.5% and 60.8% above the strongest prior baselines, respectively), at an average cost of $15.30 and 57 minutes per project. When translating tests, ReCodeAgent achieves 99.3% assertion equivalence, 0.91 cosine similarity, and 94.9% assertion type match, establishing the practicality of the "translate tests alongside code" design. An ablation study shows that removing the Analyzer, Planning, or Validator agent reduces the test pass rate by 22.7%, 25.3%, or 30.3%, respectively, while increasing trajectory complexity by 28% under the process-centric metrics of Liu et al. [39].

**C5. Open artifacts, benchmarks, and trajectory datasets for reproducibility and follow-up research.** All four research chapters are accompanied by publicly released code, data, and trajectories [40], [41], [42], [43]. These artifacts include (1) the 15-category bug taxonomy dataset of 1,748 manually labeled GPT-4 buggy translations, (2) the ten Java-to-Python AlphaTrans translation benchmarks and their human-repaired "ground-truth" counterparts, (3) the 2,219-pair MatchFixAgent cross-PL validation benchmark spanning six PL pairs, and (4) the ReCodeAgent trajectories over 118 projects and four PL pairs. Together these artifacts form one of the largest publicly available collections of repository-level code translation data, and they have begun to enable downstream research on repair, benchmark construction, and agentic workflow analysis.

### 1.2 Enduring Takeaways

The specific LLMs, benchmarks, and headline success rates reported in this dissertation will change as models and tooling evolve. Two concepts, however, are likely to remain relevant for repository-level code translation in the years ahead.

First, *validation through observable behavior* in both the source and target PL. Across AlphaTrans, MatchFixAgent, and ReCodeAgent, correctness is established by executing translated code and comparing outcomes against the source on matched inputs—not by lexical similarity or compilation alone. When a test constructs a complex object, validation must ensure that the corresponding object in the target PL exposes the same attributes, invariants, and observable interactions; a translation that compiles but returns a structurally different object remains incorrect. This criterion is anchored in program semantics rather than in any particular model, and therefore survives improvements in LLM fluency: stronger models may produce more idiomatic code, but idiomatic code that mis-maps a library API or silently weakens an assertion still fails until observable behavior aligns across languages.

Second, *principled multi-agent decomposition* of the translation workflow. Repository-level translation couples analysis, planning, co-translation of code and tests, and iterative validation over tens of thousands of lines and cross-file dependencies. Generating an entire repository from scratch in a single agentic pass is likely to remain challenging even as models grow more capable, because the task is long-horizon, mixes conflicting objectives, and demands sustained reasoning about project structure. The division of labor embodied in ReCodeAgent—dedicated agent scaffolds for each phase, empirically shown to outperform monolithic alternatives (Chapter 7)—addresses structural properties of the problem rather than limitations of a particular model generation, and is therefore expected to outlast the systems evaluated here.

### 1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 surveys the state-of-the-art in code translation, repository-level code translation and validation, translation validation and repair, and LLM agents, and identifies four concrete research gaps that the rest of this dissertation addresses. Chapter 3 formalizes the repository-level code translation problem as the construction of a target repository that preserves the observable semantics of the source repository under a given set of correctness criteria, decomposes it into four sub-problems, and articulates four research hypotheses (H1–H4) along with a unifying thesis statement. Chapter 4 presents the first large-scale empirical study of LLM-based code translation, covering 43,379 translations across seven LLMs and five PLs on both crafted benchmarks and real-world projects, and derives a 15-category, 5-group taxonomy of LLM translation bugs through manual labeling of 1,748 buggy GPT-4 translations, thereby establishing Hypothesis H1. Chapter 5 introduces AlphaTrans, the first neuro-symbolic compositional pipeline for translating and validating entire real-world repositories, and evaluates it on ten real-world Java-to-Python projects together with a human-subject study that quantifies post-pipeline developer effort, establishing Hypothesis H2. Chapter 6 introduces MatchFixAgent, a language-agnostic, multi-agent framework for repository-level translation validation and repair that combines approximate semantic analyses with a test generator–repair agent and a verdict agent, and evaluates it against four prior techniques on 2,219 source–translation pairs across six PL pairs, establishing Hypothesis H3. Chapter 7 introduces ReCodeAgent, the first end-to-end, fully agentic, PL-agnostic framework for repository-level code translation and validation, and evaluates it against four prior techniques on 4,583 translation units drawn from 118 real-world projects across four PL pairs, establishing Hypothesis H4. Finally, Chapter 8 summarizes the dissertation's contributions, reflects on the lessons learned, and outlines directions for future work.

The four research chapters can be read independently: Chapter 4 is a self-contained empirical study; Chapter 5 is a self-contained technique paper; Chapter 6 and Chapter 7 each present a self-contained agentic system. Readers interested only in the dissertation's highest-level argument may read Chapter 1–Chapter 3 and Chapter 8, while readers interested in a specific technical artifact should consult the corresponding research chapter.

The research presented in this dissertation has been published at the following venues:

- Rangeet Pan*, **Ali Reza Ibrahimzada***, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand, "Lost in Translation: A Study of Bugs Introduced by Large Language Models While Translating Code", in proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE'24), April 14–20, 2024, Lisbon, Portugal [22].

- **Ali Reza Ibrahimzada**, "Program Decomposition and Translation with Static Analysis", in proceedings of the 46th IEEE/ACM International Conference on Software Engineering Companion (ICSE'24 Companion), April 14–20, 2024, Lisbon, Portugal. [**Ranked 3rd in IEEE/ACM Student Research Competition at ICSE'24**] [44].

- **Ali Reza Ibrahimzada**, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand, "AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation", in proceedings of the ACM Conference on Foundations of Software

Engineering (FSE'25), June 23–27, 2025, Trondheim, Norway [45].

- **<u>Ali Reza Ibrahimzada</u>**, Brandon Paulsen, Reyhaneh Jabbarvand, Joey Dodds, and Daniel Kroening, "MatchFixAgent: Language-Agnostic Autonomous Repository-Level Code Translation Validation and Repair", in proceedings of the 43rd International Conference on Machine Learning (ICML'26), July 6–11, 2026, Seoul, South Korea [46].

- **<u>Ali Reza Ibrahimzada</u>**, Brandon Paulsen, Daniel Kroening, and Reyhaneh Jabbarvand, "ReCodeAgent: A Multi-Agent Workflow for Language-Agnostic Translation and Validation of Large-Scale Repositories", currently under submission [47].

In addition, the following publications are done during PhD but are not included in the dissertation:

- **<u>Ali Reza Ibrahimzada</u>**, Yang Chen, Ryan Rong, and Reyhaneh Jabbarvand, "Challenging Bug Prediction and Repair Models with Synthetic Bugs", in proceedings of the 25th IEEE International Conference on Source Code Analysis & Manipulation (SCAM'25), September 7–12, 2025, Auckland, New Zealand [48].

- **<u>Ali Reza Ibrahimzada</u>**, Yigit Varli, Dilara Tekinoglu, and Reyhaneh Jabbarvand, "Perfect Is the Enemy of Test Oracle", in proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'22), November 14–18, 2022, Singapore, Singapore [49].

- Kaiyao Ke, **<u>Ali Reza Ibrahimzada</u>**, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand, "Advancing Automated In-Isolation Validation in Repository-Level Code Translation", currently under submission [50].

# Chapter 2 Background and Related Work

This chapter provides an overview of the related research on automated code translation, repository-level translation and validation, and LLM agents for software engineering. Furthermore, it discusses the research gap and positions the techniques proposed in this dissertation among the body of literature. To that end, the related work is organized into five complementary categories spanning non-LLM-based code translation, LLM-based code translation, repository-level code translation, translation validation and repair, and agentic software engineering systems, followed by four concrete research questions that delineate the unaddressed portion of this landscape and motivate Chapter 4–Chapter 7.

## 2.1 Related Work

The work in this dissertation sits at the intersection of five complementary research areas: non-LLM-based code translation, LLM-based code translation, repository-level code translation, translation validation and repair, and LLM agents for software engineering. This section surveys each area in turn, with the aim of placing the contributions of Chapter 4–Chapter 7 precisely within the existing landscape. Chapter-specific related-work discussions (§4.9, §5.8, §6.5, and §7.5) refine the comparison with the closest prior state-of-the-art for each individual technique; the present section provides the dissertation-wide context.

### 2.1.1 Non-LLM-Based Code Translation

Before the LLM era, automated code translation was dominated by two approaches: rule-based transpilers and statistical or neural machine translation over source code.

Rule-based transpilers are hand-engineered programs that apply syntactic and semantic rewrite rules to convert code from a source PL to a target PL. Among the most widely used are C2Rust [18], which translates C to Rust (typically with heavy use of `unsafe` blocks), CxGo [19], which translates C to Go, Sharpen [20], which translates Java to C#, and Java2CSharp [21], which targets the same pair. Because every rewrite rule must be crafted manually, transpilers are inherently PL pair-specific; extending a transpiler to a new source or target PL requires re-engineering the entire rule set. Prior studies have also shown that transpiler output frequently lacks naturalness and idiomaticity in the target PL, forcing developers to spend significant post-translation effort cleaning up the generated code [22], [23]. Rule-based transpilers excel at preserving syntactic structure, but they struggle with PLs whose semantics or programming paradigms differ substantially from the source (e.g., object-oriented to functional, manual to garbage-collected memory), and they are largely incapable of reasoning about library API mismatches between the two PLs.

A second family of techniques treats source code as a sequence of tokens and applies phrase-based or tree-based statistical machine translation (SMT) to the translation problem. Nguyen *et al.* [24], [25], [26] explored lexical and syntactic SMT, primarily for Java-to-C#; Chen *et al.* [27] introduced tree-to-tree neural networks that preserve the AST structure of the source; and Karaivanov *et al.* [28] studied phrase-based SMT for C#-to-Java. SMT-for-code approaches produced translations that were often more natural than rule-based output, but they were trained on small, curated corpora of aligned programs and struggled with translations that required any non-local reasoning about the source PL. Building on this line of work, Rozière *et al.*'s TransCoder [29] established the feasibility of unsupervised code translation by leveraging the similarities between programming languages, and DOBF [31] together with follow-up work [30] used de-obfuscation pre-training to improve cross-PL transfer. While these techniques produced translations that were idiomatic and typically compilable, they were evaluated on *function-level* translation of crafted benchmarks (e.g., GeeksforGeeks [29]) and were not designed for repository-level use cases. Unlike all of the above, the techniques in Chapter 5–Chapter 7 target real-world repositories with cross-file dependencies and developer-written test suites.

### 2.1.2 LLM-Based Code Translation

The rise of general-purpose LLMs such as GPT [32] and Claude [33], and of code-specialized LLMs such as StarCoder [34], CodeGen [35], CodeGeeX [36], SantaCoder [51], PolyCoder [52], CodeT5 [53], BLOOM [54], Llama 2 [55], and DeepSeek-Coder [37], ushered in a new era in which code translation could be performed as a zero- or few-shot prompting task. A large body of work has investigated the effectiveness of LLM-based translation on crafted benchmarks [56], [57], [58], [59], [60], [61], [62], [63], [64]. The consistent finding is that LLMs outperform transpilers and SMT baselines on small programs, but their effectiveness collapses on real-world projects, and the causes of that collapse are not well understood.

The empirical study that opens this dissertation, "Lost in Translation" [22] (Chapter 4), was designed to fill exactly this gap. It (a) evaluates seven LLMs (including both general and code LLMs), (b) covers five PLs over three crafted benchmarks and two real-world open-source projects (Apache Commons CLI [65] and Python Click [66]), (c) includes EvalPlus [67] alongside CodeNet and AVATAR, (d) performs a 630 person-hour manual labeling of 1,748 buggy GPT-4 translations to produce a 15-category taxonomy of translation bugs, (e) compares LLM-based translation head-to-head against non-LLM transpilers (C2Rust, CxGo, Java2CSharp), and (f) evaluates iterative prompt engineering as a mitigation. Chapter 4 thereby provides the first published taxonomy of translation bugs and the first systematic evidence that LLM-based translation quality is substantially worse on real-world code than on crafted benchmarks, and it furnishes the failure mode vocabulary that the remaining chapters build upon.

A parallel line of LLM-based research has investigated the use of LLMs not for *introducing* bugs through code generation but for *repairing* them through automated program repair, including infilling-style [68], conversational [69], and retrieval-augmented [70] approaches as well as CodeBERT-based fault localization [71] and broader practical repair systems [72], [73], [74]. The bug taxonomy of Chapter 4 was constructed in part to bridge these two literatures: by exposing the structure of the bugs that LLMs introduce during translation, it gives downstream LLM-based repair techniques a concrete vocabulary of failure modes to target. The compositional translation pipeline of Chapter 5 and the agentic repair pipelines of Chapter 6 and Chapter 7 can be seen as specialized program repair systems tailored to the structure of these translation bugs.

### 2.1.3 Repository-Level Code Translation

A small but rapidly growing body of work has moved beyond function-level translation to the repository-level. SYZYGY [75] translates C to Rust using GPT-4 and an I/O-based equivalence check, instrumenting source programs to capture expected outputs. OXIDIZER [76] translates Go to Rust using language feature mapping and type-driven techniques, and validates translations by capturing and replaying inputs in the target PL. SKEL [77] translates Python to JavaScript with a minimal skeleton-based approach, validating by string-replacing Python tests into JavaScript; the technique succeeds because source Python tests are typically simple function calls and value comparisons. C2SAFERRUST [78] and a series of concurrent C-to-Rust systems [79], [80], [81], [82], [83] aim to produce safer, more idiomatic Rust translations by combining transpiler output with LLM rewriting or reinforcement learning-based refinement. Nitin et al. [84] and Yang et al. [85] have pursued formal methods-based validation of translated programs, though at the cost of scalability. Yang et al. [86] and Dehghan et al. [87] investigate the use of tests to assist LLM-driven translation, and Ou et al. [88] studies the practical challenges of repository-level translation in an industrial setting. Xue et al. [89] and Wang et al. [90] evaluate class-level and efficiency-aware translation, respectively.

ALPHATRANS [45], the subject of Chapter 5, is the first technique designed for end-to-end translation of real-world repositories *including tests*, and the first to use language interoperability (via GraalVM [91]) for in-isolation validation of translated fragments while the rest of the project remains in the source PL. A recurring limitation of this entire family of work—including AlphaTrans—is that each technique supports only a *single* source–target PL pair, because the PL-specific program analysis, dependency extraction, and interoperability runtime each require substantial engineering effort: AlphaTrans is roughly 11,000 lines of code, Oxidizer roughly 19,000, Skel roughly 4,000, and C2Rust more than 100,000. Addressing this quadratic engineering cost is the explicit goal of Chapter 6 and Chapter 7.

### 2.1.4 Translation Validation and Repair

Even a syntactically valid translation is useless if it does not preserve the semantics of the source; validating the translation is therefore as important as producing it. Prior work has explored three broad families of validation techniques: test execution-based validation, formal methods-based validation, and fuzzing-based validation.

The dominant approach reuses the test suite of the source project. Techniques such as AlphaTrans [45], Oxidizer [76], Skel [77], Syzygy [75], and the work of Ou et al. [88] either translate the tests into the target PL and execute them, or use language-interoperability mechanisms such as GraalVM [91] to execute source PL tests against target PL translations [92]. Test execution-based validation is sound in principle but incomplete: developer tests are often inadequate [50], [93], and a passing test suite does not imply equivalence. Moreover, language-interoperability runtimes typically exist for only a handful of PL pairs. Yang et al. [85] and Nitin et al. [84] circumvent this incompleteness by applying formal verification and specification inference, respectively, to establish equivalence between source and translated programs. Formal techniques offer stronger guarantees than test execution but at the cost of scalability: they are typically evaluated on crafted benchmarks and do not scale to entire real-world repositories. Eniser et al. [94] takes a third path and uses differential fuzzing to expose divergent behaviors between source and translated programs. Fuzzing-based validation is effective on programs with simple input types but struggles to generate valid inputs for complex, deeply typed real-world code [95].

When validation determines a translation as inequivalent, existing repository-level techniques typically repair the incorrect translation by re-prompting the LLM with the failing test output as additional context [22], [45], [75], [76], [86], [88], or by delegating repair to a human in the loop [77]. Re-prompting has proven only modestly effective [45], and human-in-the-loop repair clearly does not scale. Unlike all of these techniques, MatchFixAgent (Chapter 6) replaces both approaches with an LLM agent that combines approximate program analyses with targeted test generation, and demonstrates that this combination repairs 32.1% more translation bugs than the strongest re-prompting baseline while requiring only roughly 280 lines of code to support a new PL pair.

### 2.1.5 LLM Agents for Software Engineering

The final thread of related work concerns LLM agents—systems in which one or more LLMs interleave reasoning, tool use, and observation in the ReAct [96] style. Benchmarks such as SWE-bench [97], [98], [99] have catalyzed the development of agentic systems for software engineering, most notably SWE-agent [100], AutoCodeRover [101], SpecRover [102], Agentless [103], OpenHands [104], RepairAgent [105], SWE-Search [106], and commercial assistants such as Amazon Q [107] and Devin [108]. These systems have demonstrated that LLM agents can resolve real-world software issues, although the best reported success rates on SWE-bench remain well below human performance.

Most existing agentic systems target bug fixing or feature addition, not translation. A small number of agentic systems target code translation specifically: CRUST-Bench [109] evaluates agents for C-to-Rust translation, RepoTransAgent [110] explores agents for cross-repository translation tasks, Li et al. [111] investigates adversarial evaluation of translation agents, and Sim et al. [112] studies large-scale translation. A consistent limitation of these systems, analyzed in detail in Chapter 7, is that they either rely on PL-specific tooling (thereby forfeiting the language-agnosticism that agents promise) or rely only on naive, PL-agnostic tools such as bash commands (thereby forfeiting the semantic grounding that makes neuro-symbolic pipelines effective). ReCodeAgent bridges this gap by grounding each of its four agents in PL-agnostic semantic tooling—specifically Tree-sitter [113] and Language Server Protocols [114]—thereby preserving both agentic autonomy and semantic precision.

The multi-agent architecture of MatchFixAgent and ReCodeAgent is further motivated by a rapidly growing literature on role separation in LLM agents. Studies have shown that a single agent asked to both produce and judge code is susceptible to reward hacking (e.g., weakening tests to obtain a passing verdict) and to conflicting-objective bias [115], [116], [117], [118], [119], [120], [121], [122], [123], [124], [125]. Separating translation from validation into distinct agents is therefore not merely an engineering convenience but a principled response to a known failure mode of single-agent pipelines.

## 2.2 Research Gap

In this section, I identify the research gap in the related literature by answering the following research questions:

- *Generalizability across PL pairs.* To what extent do prior code translation techniques generalize across the quadratic space of source–target PL pairs, and do prior empirical evaluations measure that generalizability with *functional* (Pass@$K$-style) or merely *non-functional* (CodeBLEU-style) metrics?
- *In-depth analysis of LLM translation failures.* What types of bugs do LLMs introduce when translating code, and have prior studies characterized those failures precisely enough—on real-world software—to inform downstream tool designing?
- *Coverage of repository-level translation and validation.* Can prior pipelines

translate *and* validate complete real-world repositories, including their tests, through a unified translate-and-validate workflow, rather than through a translate-first-validate-next architecture that is fundamentally incompatible with repository-level code?

- *Agentic systems for repository-level translation, validation, and repair.* Do prior agentic systems combine PL-agnosticism with semantic grounding for end-to-end repository-level translation and validation, and are they evaluated transparently along process-centric and cost dimensions?

### 2.2.1 Generalizability Across PL Pairs

Three independent threads in prior work share a single root cause: the cost of supporting a new PL pair has historically been borne by hand, either inside the translator or inside the evaluation harness.

First, rule-based transpilers such as C2Rust [18], Java2CSharp [21], CxGo [19], and Sharpen [20] apply syntactic and semantic rewrite rules that are crafted manually for a single source–target pair; C2Rust alone exceeds 100,000 lines of code. Because every rewrite rule is PL-specific, extending such a transpiler to a new source or target language requires re-engineering the entire rule set, and the resulting translations are widely reported to be non-idiomatic and to lean heavily on `unsafe` constructs [22], [23]. Second, the first generation of repository-level LLM-based pipelines—Syzygy [75], Oxidizer [76], Skel [77], and AlphaTrans [45] included—inherits this cost model: each system is pinned to a single source–target pair because its program analysis, dependency extraction, and interoperability runtime are written directly against language-specific toolchains, with implementation sizes of roughly 11,000 lines for AlphaTrans, 19,000 for Oxidizer, and 4,000 for Skel. Because the set of interesting PL pairs is quadratic in the number of PLs, this engineering model does not scale, and even concurrent and follow-up systems (C2SaferRust [78], RustMap [79], EvoC2Rust [81]) inherit it. Third, the empirical literature surrounding these systems has historically reported translation quality using non-functional, surface-level metrics—CodeBLEU [126], CrystalBLEU [127], exact match, syntax match, and data-flow match—on which LLMs can score highly while still producing code that fails to compile or execute [128], [129]. Because these metrics do not measure semantic preservation, they cannot certify that an LLM *actually* generalizes across PL pairs, only that it produces lexically plausible target PL output.

There is thus a need for a comprehensive empirical investigation that (i) measures LLM translation quality with *functional*, execution-based metrics (compilation, runtime, and test execution), (ii) covers many PLs and PL pairs simultaneously rather than just one, and (iii) compares LLMs head-to-head against PL-specific transpilers on the same projects, so that the question of whether LLM-based translation is intrinsically more general than rule-based translation can be settled empirically rather than asserted. Chapter 4 closes this gap with 43,379 translation pairs spanning seven LLMs (GPT-4, Llama 2, TheBloke-Vicuna, TheBloke-Airoboros, StarCoder, CodeGeeX, and CodeGen), five PLs (C, C++, Go, Java, Python), three crafted benchmarks (CodeNet [130], AVATAR [128], EvalPlus [67]), and two real-world projects (Apache Commons CLI [65], Python Click [66]). Every translation is evaluated against existing tests rather than non-functional metrics, and the study includes a head-to-head comparison with C2Rust, CxGo, and Java2CSharp on the same benchmarks. The engineering cost half of this generalizability problem—reducing the per PL-pair effort required by the pipelines themselves—is taken up later in Chapter 6 and Chapter 7.

### 2.2.2 In-Depth Analysis of LLM Translation Failures

Prior to this dissertation, empirical studies of LLM-based code translation [56], [57], [58], [59], [60], [61], [62], [63], [64] reported aggregate success rates on crafted benchmarks but did not analyze the *root causes* of failure on real-world software, nor did they manually label bug instances at scale. As a result, neither researchers nor practitioners possessed a shared vocabulary of failure modes against which downstream tooling could be designed. The methodological tradition of cataloging the bugs introduced by automated code generation tools is long-standing in software engineering [131] and has recently been extended to bugs found inside deep learning models themselves [132], [133], [134], but no prior work had applied this methodology to the bugs that LLMs *introduce* during code translation.

There is thus a need for a systematic, taxonomy-driven empirical understanding of LLM translation failures on real-world code, grounded in large-scale manual labeling and accompanied by a quantitative evaluation of practical mitigations. Chapter 4 closes this gap through 630 person-hours of manual labeling, by eight labelers, of 1,748 buggy GPT-4 translations, yielding a 15-category, 5-group taxonomy of LLM translation bugs that generalizes across models, PLs, and benchmarks. The same chapter quantifies iterative prompt engineering as a mitigation strategy, showing an average improvement of 5.5% across the studied LLMs (and 12% for GPT-4, the strongest). The resulting taxonomy is the first published catalogue of LLM translation bugs and is used as a debugging lens by every subsequent chapter of this dissertation.

### 2.2.3 Coverage of Repository-Level Translation and Validation

Even with the failure modes of LLM-based translation catalogued, applying LLMs at the scale of a real-world repository raises challenges that function-level studies cannot expose. Real-world projects routinely span tens or hundreds of thousands of lines, contain cross-file dependencies, and rely on third-party libraries whose APIs have no direct counterpart in the target PL. The entire project—and frequently even a single class—does not fit into the context window of current LLMs [44], and even when relevant code is included verbatim, the short and non-uniform attention of current LLMs [38] prevents them from reasoning reliably about inter-procedural dependencies. The natural response, suggested by program analysis research, is to *decompose* the repository into translatable fragments and translate them in dependency order rather than asking the LLM to ingest the project whole.

Decomposition alone, however, is insufficient: most prior repository-level pipelines [75], [76], [77], [84], [85], [86], [88], [94] adopt a “translate first, validate next” architecture in which every fragment is translated in one pass and the entire translated repository is validated only afterwards. This architecture forfeits the opportunity to use validation outcomes as feedback for the next translation step and, more critically, founders on the *test-coupling* problem [45], in which the endpoints of long call chains cannot be executed in the target PL until the entire chain has been translated. The architectural barrier of prior neuro-symbolic pipelines is therefore twofold: (a) they treat translation and validation as sequential phases rather than as a single per-fragment step, and (b) they translate the repository’s *code* without translating its *tests*, leaving validation dependent on either incomplete language interoperability runtimes or on later manual effort. The consequence is that prior systems either translate function-level fragments without integrating them into a runnable whole, translate small or toy projects whose source files fit in a single context window, or translate a repository but validate only partially (e.g., checking compilation but not test execution).

There is thus a need for a unified, compositional translation-and-validation pipeline that (i) decomposes the source project into syntactic fragments via static analysis, (ii) translates and validates each fragment as a single step in reverse call-graph order, (iii) translates the repository’s tests alongside its code, and (iv) validates each translated fragment *in isolation* against the rest of the project written in the source PL. Chapter 5 closes this gap through AlphaTrans, a neuro-symbolic compositional pipeline that decomposes the source project into fragments using static analysis, translates them in reverse call order, and uses GraalVM-based language interoperability [91] for in-isolation fragment validation while the remainder of the project remains in the source PL. AlphaTrans is the first technique to translate an entire repository—code *and* tests—end-to-end and the first to leverage language interoperability for in-isolation fragment validation.

### 2.2.4 Agentic Systems for Repository-Level Translation, Validation, and Repair

A unified compositional pipeline such as AlphaTrans solves the repository-level problem for a *fixed* PL pair, but it leaves two further problems open. First, the engineering cost identified in §2.2.1 recurs at *every* stage of the pipeline: the validation, repair, and test generation components of a system like AlphaTrans must be re-implemented from scratch for each new PL pair, and the same is true of concurrent and follow-up systems such as C2SaferRust [78], RustMap [79], and EvoC2Rust [81]. Second, the small but growing body of agentic translation systems either employs only naive, PL-agnostic tools such as bash commands without semantic grounding [109], employs PL-specific semantic tools that re-introduce the engineering cost [110], [111], or focuses on non-translation tasks entirely [100], [101], [102], [103], [104], [105], [106], [135]. None delivers *both* language-agnosticism *and* semantic grounding for repository-level translation *and* validation. Worse, agentic translation systems tend to be evaluated only by outcome-only metrics that conflate several known threats to validity (assertion deletion during test translation, generation of tests with no-op assertions, claims of "passing" for projects with inadequate test suites [50]) and rarely report trajectories, tool execution counts, or monetary cost in a form that downstream researchers can analyze. Without process-centric metrics such as those of Liu et al. [39]—node count, tool execution count, and trajectory length—it is impossible to compare agentic systems in a principled way.

There is thus a need for agentic translation-and-validation frameworks that (i) ground their reasoning in PL-agnostic semantic tooling so that engineering effort is decoupled from the choice of PL pair, (ii) separate translation from validation into distinct agents to avoid known single-agent reward-hacking failure modes [115], [116], [121], [122], and

(iii) are evaluated transparently along process-centric and cost dimensions in addition to outcome-only metrics. This gap is closed jointly by Chapter 6 and Chapter 7. MatchFixAgent (Chapter 6) introduces a multi-agent validation-and-repair framework that grounds its decisions in approximate, PL-agnostic semantic analyses—control flow, data flow, library APIs, exception handling, and specifications—and that requires only roughly 280 lines of PL-specific code per PL pair, repairing 32.1% more translation bugs than the strongest prior re-prompting baseline while reporting ablations that isolate the contribution of each component. ReCodeAgent (Chapter 7) extends the PL-agnostic principle to the *translation* stage itself: it introduces the first four agent (*Analyzer*, *Planning*, *Translator*, *Validator*) framework in which every agent is grounded in PL-agnostic tooling—Tree-sitter [113] and Language Server Protocols [114]—and which reports per PL-pair compilation, test pass, assertion fidelity, trajectory complexity, and monetary cost metrics across four PL pairs (C–Rust, Go–Rust, Java–Python, Python–JavaScript) spanning six PLs.

# Chapter 3 Research Problem

The literature surveyed in Chapter 2 establishes that prior work falls short of automatically translating and validating real-world repositories across arbitrary PL pairs. Rule-based transpilers and statistical translation systems do not generalize beyond a handful of language pairs, do not produce idiomatic target PL code, and do not cope with the cross-file dependencies typical of real codebases. LLM-based techniques translate idiomatically but—as the empirical study reported in Chapter 4 demonstrates and as the wider community has begun to corroborate—collapse on real-world projects in ways that the literature has not previously characterized. The small body of repository-level techniques developed in parallel with this dissertation either remain tied to a single source–target PL pair (AlphaTrans [45], Oxidizer [76], Skel [77], Syzygy [75]) or validate only partial artifacts (compilation but not test execution, single fragment translation rather than whole-repository translation including tests). And the small body of agentic translation systems either rely on PL-specific tooling (forfeiting the language-agnosticism that agents promise) or rely only on naive, PL-agnostic tools such as bash commands (forfeiting the semantic grounding that makes neuro-symbolic pipelines effective).

Closing these gaps is the goal of this dissertation, but the path is not a single technical leap. It is, instead, a sequence of four interrelated challenges, each of which yields one of the four research chapters:

1. There is a lack of empirical understanding of *what* goes wrong when LLMs translate code in real-world settings, and consequently no shared vocabulary of failure modes that downstream tooling can target. Existing studies of LLM-based code translation focus overwhelmingly on crafted benchmarks, do not perform manual labeling of bug instances at scale, and do not compare LLMs head-to-head against non-LLM transpilers on the same projects. *Thereby, there is a need for a*

*large-scale, multi-LLM, multi-PL empirical study of LLM-based code translation, accompanied by a manually labeled bug taxonomy and a quantitative evaluation of practical mitigations such as iterative prompt engineering.*

2. Even when the failure modes are catalogued, prior repository-level translation techniques do not translate *and* validate complete real-world repositories: they translate function-level fragments without integrating them into a runnable whole, translate only small or toy projects whose source files fit in a single LLM context window, or translate code without translating the accompanying tests. Long call chains and inter-procedural dependencies—the "test coupling" problem—make repository-level validation particularly challenging, because the endpoints of those chains cannot be independently executed in the target PL until the entire chain has been translated. *Thereby, there is a need for a compositional translation pipeline that decomposes a repository into syntactic fragments, translates them in dependency order, and validates each fragment in isolation against the rest of the project written in the source PL.*

3. Once a single PL pair has been solved, the engineering cost of solving each additional pair quickly becomes prohibitive. Existing repository-level pipelines—AlphaTrans at roughly 11,000 lines of code, Oxidizer at roughly 19,000, Skel at roughly 4,000, and the rule-based C2Rust at more than 100,000—scale poorly across the quadratic space of source–target PL pairs because their program analysis, dependency extraction, and validation runtimes are all written directly against language-specific toolchains. Crucially, this engineering cost recurs at every pipeline stage: validation, repair, test generation, and so on. *Thereby, there is a need for a validation-and-repair framework that grounds its decisions in approximate, language-agnostic semantic analyses behind a multi-agent architecture, so that adding a new PL pair requires only a small constant amount of*

*PL-specific engineering rather than a full rewrite.*

4. Finally, even a generalizable validation-and-repair framework leaves untouched the harder question of generalizing the *translation* stage itself. Prior agentic translation systems either rely on PL-specific tooling, rely only on naive PL-agnostic tools such as bash commands, or focus on tasks other than translation entirely; none combines language-agnosticism with semantic grounding for repository-level translation *and* validation, and few report the trajectories, tool-use counts, or monetary costs that would allow principled comparison with prior work. *Thereby, there is a need for a multi-agent, end-to-end pipeline whose every agent is grounded in PL-agnostic semantic tooling and whose performance is reported transparently along compilation, test-pass, assertion-fidelity, trajectory-complexity, and monetary-cost dimensions.*

The remainder of this chapter formalizes the problem these four challenges jointly address (§3.1), states the four research hypotheses that drive the dissertation (§3.2), and distills the central claim of the dissertation into a single thesis statement (§3.3).

## 3.1 Problem Statement

The four challenges enumerated above can be unified under a single research problem, which I refer to throughout this dissertation as the *repository-level code translation problem.* A source project $P_s = (F_s, T_s, D_s)$ consists of source functions $F_s = \{f_s^1, \dots, f_s^n\}$, a developer-written test suite $T_s$, and a set of third-party dependencies $D_s$, all written in a source programming language $L_s$. Given a target programming language $L_t$, the goal is to produce a target project $P_t = (F_t, T_t, D_t)$ written in $L_t$ such that:

1. **Buildability.** $P_t$ compiles without errors against the resolved dependency set $D_t$, where each $d_t \in D_t$ is either an idiomatic counterpart in $L_t$ of some $d_s \in D_s$ or a runtime equivalent dependency synthesized when no direct counterpart exists.

2. **Functional equivalence.** For every translated function $f_t^i \in F_t$ and its source PL counterpart $f_s^i \in F_s$,

$$\forall \vec{x}_s,\ \forall \vec{x}_t :\ \vec{x}_s \mapsto \vec{x}_t \implies [\![ f_s^i ]\!](\vec{x}_s) \simeq [\![ f_t^i ]\!](\vec{x}_t),$$

   where $\vec{x}_s$ and $\vec{x}_t$ denote function inputs in $L_s$ and $L_t$, $\mapsto$ is a mapping of concrete values from $L_s$ to $L_t$, $[\![\cdot]\!]$ denotes semantic interpretation in the respective language, and $\simeq$ denotes observational equivalence (equality of return values and side effects on the assertions exercised by the test suite).

3. **Test suite preservation.** For every test $t_s \in T_s$ that passes when executed against $P_s$, the translated test $t_t \in T_t$ exists, exercises the assertions of $t_s$ under the mapping $\mapsto$, and passes when executed against $P_t$.

The functional equivalence condition above is, in general, undecidable for arbitrary $L_s, L_t$, and arbitrary function pairs. Following the standard practice in the validation literature [45], [76], [84], [85], [94], this dissertation approximates condition (2) by restricting the universal quantifier over inputs to the inputs actually exercised by $T_s$, so that condition (3) operationally subsumes condition (2) on the part of the input space that the developer test suite covers. The remaining input space is addressed by the test generation components introduced in Chapter 6 and Chapter 7, which extend $T_s$ to expose divergences that the original suite does not exercise. Crucially, this formulation differs from function-level translation in that the unit of translation is the entire project $P_s$, including its tests $T_s$ and its dependency set $D_s$, and the unit of validation is the end-to-end execution of $T_t$ against $P_t$.

The repository-level translation problem decomposes naturally into four sub-problems that mirror the four challenges of Chapter 3. The first is the *characterization sub-problem*: identifying and quantifying the failure modes that arise when an LLM is applied directly to the translation of $f_s^i \in F_s$, so that downstream tooling can target them. The second is the *single-pair compositional sub-problem*: producing $P_t$ for a fixed $\langle L_s, L_t \rangle$ by decomposing $P_s$ into syntactic fragments, translating them in dependency order, and validating each fragment in isolation against the rest of $P_s$ that has not yet been translated. The third is the *cross-pair validation-and-repair sub-problem*: validating and repairing $\langle f_s^i, f_t^i \rangle$ pairs across an arbitrary number of $\langle L_s, L_t \rangle$ pairs at engineering cost that is sublinear—ideally constant—in the size of the target language toolchain. The fourth is the *cross-pair end-to-end sub-problem*: extending the same PL-agnostic principles to the production of $F_t$, $T_t$, and $D_t$ themselves, so that the entire translation-and-validation pipeline operates on arbitrary $\langle L_s, L_t \rangle$ pairs without per-pair re-engineering.

A solution to the overall problem must satisfy a set of measurable success criteria that the four sub-problems impose jointly. Specifically, it must achieve high *compilation success rates* on $P_t$, high *test pass rates* on $T_t$, high *assertion fidelity* between $T_s$ and $T_t$ (so that a passing $t_t \in T_t$ exercises the same observable behavior as its counterpart $t_s \in T_s$), *small per PL-pair engineering cost* measured in lines of $L_s$- or $L_t$-specific code, and *practical operational cost* measured in monetary spend per project and wall-clock time per project. Beyond these outcome metrics, a principled solution must also expose its internal trajectory—node count, tool execution count, and trajectory length in the sense of Liu et al. [39]—so that its behavior can be compared against future systems with the same level of rigor that this dissertation applies to prior work. The chapters that follow contribute techniques and empirical evidence that, together, satisfy these criteria for the four PL pairs studied (Java–Python, C–Rust, Go–Rust, and Python–JavaScript) while pointing toward the generalization of the same principles to the broader quadratic space of PL pairs.

## 3.2 Research Hypothesis

This dissertation investigates the following four hypotheses, each corresponding to one of the four challenges of Chapter 3 and each addressed in detail by one of the four research chapters that follow.

- Empirical bug studies have a long tradition in software engineering [131] and have recently been extended to deep learning models themselves [132], [133], [134], but no prior work has applied this methodology to the bugs that LLMs *introduce* during code translation on real-world projects. The literature has documented in aggregate that LLMs translate crafted benchmarks well and real-world projects poorly, but it has not characterized the failure modes precisely enough to inform downstream tooling, has not labeled bug instances at scale, and has not compared LLM translation head-to-head against rule-based transpilers on the same benchmarks. Closing this gap is a prerequisite for every subsequent technical contribution: without a vocabulary of failure modes, neither this dissertation nor downstream researchers can design pipelines that target the right failures. Thereby, I conjecture that a sufficiently large empirical study, accompanied by manual labeling of representative buggy translations and by a head-to-head comparison with non-LLM transpilers, can produce a generalizable taxonomy of LLM translation failures and quantify the effectiveness of practical mitigations such as iterative prompt engineering.

  **Hypothesis 1 (Chapter 4).** *A multi-LLM, multi-PL empirical study with manual bug labeling and head-to-head transpiler comparisons can yield a generalizable taxonomy of LLM translation bugs that informs downstream repository-level translation techniques.*

- The compositional structure of real-world repositories—file hierarchies, class hierarchies, methods organized into call graphs, and tests that exercise long cross-file call chains—is precisely what defeats off-the-shelf LLM translation, but it is also precisely what classical program analysis is designed to expose. The two technologies have largely been treated separately, but the failure analysis of Chapter 4 suggests that they should be fused: program analysis to decompose the repository into translatable fragments and to determine the order in which those fragments must be translated, and LLMs to perform the actual translation of each fragment, with cross-language interoperability used to validate each translated fragment in isolation while the rest of the project remains in the source PL. Such a neuro-symbolic compositional pipeline should, in principle, scale to entire real-world repositories including their tests; the question is whether it does so in practice. Thereby, I conjecture that an LLM augmented with static analysis of the source repository and with cross-language interoperability for in-isolation fragment validation can translate complete real-world repositories at a scale and quality that prior repository-level techniques cannot reach.

**Hypothesis 2 (Chapter 5).** *A neuro-symbolic compositional pipeline that decomposes the source repository, translates fragments in reverse call graph order, and validates each fragment in isolation via cross-language interoperability can translate and validate complete real-world repositories—code and tests—between Java and Python.*

- Solving repository-level translation for a single PL pair via heavy symbolic engineering, as Hypothesis 2 proposes, comes at a price: the symbolic infrastructure—static analysis, dependency extraction, language interoperability runtime—is tied to the particular source and target PLs and must be re-implemented for every new pair. The same is true of every prior repository-level translation system. Because the set of interesting PL pairs is quadratic in the number of PLs, this engineering model does not scale, and even concurrent and follow-up systems inherit it. The validation-and-repair stage is, however, more loosely coupled to the PL than the translation stage itself: validation needs only *approximate* information about control flow, data flow, library APIs, exceptions, and specifications, all of which can be obtained from PL-agnostic primitives such as Tree-sitter parsers and Language Server Protocols. Thereby, I conjecture that an LLM agent framework whose validation and repair decisions are grounded in approximate, language-agnostic semantic analyses, and whose role separation prevents reward hacking failure modes documented in single-agent pipelines [115], [116], [121], [122], can validate and repair repository-level translations across arbitrary PL pairs at a fraction of the engineering cost of language-specific repair systems.

**Hypothesis 3 (Chapter 6).** *A multi-agent framework grounded in approximate, PL-agnostic semantic analyses—and with verdict and repair separated into distinct agents—can validate and repair repository-level translations across arbitrary PL pairs at hundreds of lines of PL-specific code per pair.*

- Once the validation-and-repair stage has been generalized across PL pairs, the natural next question is whether the same PL-agnostic principles extend to the *translation* stage itself. Generalizing translation is harder than generalizing validation because the translator agent must reason about the full structure of the source repository, plan a translation order, generate code in the target PL, and recover from its own mistakes—tasks that were previously thought to require PL-specific tooling. Recent advances in agentic frameworks [96], [97], [100], [101], [104] have shown that LLMs can interleave reasoning, tool use, and observation in ways that approximate program analysis behavior, provided the tools they invoke are powerful enough. Two such PL-agnostic toolkits—Tree-sitter for parsing and Language Server Protocols for symbol resolution and cross-references, exposed through the Model Context Protocol—now exist for essentially every mainstream PL. Thereby, I conjecture that a four agent architecture in which each agent

(Analyzer, Planning, Translator, Validator) is grounded in PL-agnostic semantic tooling, and in which the Validator agent inherits the validation-and-repair principles of Hypothesis 3, can perform the entire translation-and-validation workflow for real-world repositories across arbitrary PL pairs while exposing the process-centric metrics that allow principled comparison with prior work.

**Hypothesis 4 (Chapter 7).** *A four-agent (Analyzer, Planning, Translator, Validator) framework grounded in PL-agnostic tooling—Tree-sitter and Language Server Protocols—can translate and validate real-world repositories across multiple PL pairs, surpassing prior baselines on compilation, test pass, and assertion pass rates while transparently reporting trajectories and cost.*

### 3.3 Thesis Statement

The four hypotheses above can be distilled into a single thesis that captures the dissertation's central claim:

> *LLMs, when composed with the right program analyses—progressively loosened from a taxonomy-driven empirical foundation (Chapter 4) and heavy PL-specific symbolic infrastructure (Chapter 5) to approximate, PL-agnostic semantic tooling beneath a multi-agent architecture (Chapter 6, Chapter 7)—can substantially advance the translation and validation of real-world repositories, including their tests, across multiple source–target PL pairs at practical engineering and operational cost, while leaving residual failure modes that motivate continued research.*

The thesis advances three coupled claims substantiated by four research chapters, offering partial solutions to the translation-and-validation problem. First, common LLM translation failures stem from non-local reasoning about repository structure—such as identifier resolution and dependency order—which are better addressed through systematic enumeration and targeted tooling than general benchmarks suggest. Second, much of the necessary symbolic infrastructure can be PL-agnostic; using primitives like Tree-sitter and Language Server Protocols reduces the engineering cost of adding new language pairs compared to prior PL-specific approaches. Third, using the agent as a unit of pipeline composition—separating Analyzer, Planning, Translator, and Validator roles—aligns with established LLM design principles and enables independent ablation and debugging, though success rates remain below safety-critical requirements.

The chapters that follow develop the techniques (AlphaTrans, MatchFixAgent, ReCodeAgent), the empirical evidence (a 43,379 translation pair and multi-LLM study, a 15-category bug taxonomy derived from 630 person-hours of manual labeling, and evaluations across more than 150 real-world projects spanning up to six PL pairs and totaling more than 1.1M lines of code), and the publicly released artifacts that, taken together, support this thesis while exposing concrete open problems that Chapter 8 catalogues as directions for future work.

# Chapter 4 Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code

## 4.1 Introduction

Code translation, source-to-source compilation, or transpilation, entails transforming a piece of code from one programming language (PL) to another, while preserving the original functionality. Code translation has many use cases, such as modernizing enterprise applications [12], [14], [15], [16], [17], migrating legacy software in proprietary PLs to cloud-native applications implemented in general-purpose PLs [6], [7], [8], [9], [10], [11], [13], and facilitating the training of models for better code synthesis [136], [137], [138], [139]. Translating the software/code to a modern PL can significantly reduce maintenance effort, improve overall reliability, and boost non-functional properties such as security and performance [4], [140], [141], [142], [143], [144].

Due to the importance and benefits of code translation, several techniques have been developed to automate reliable translation between different PLs [18], [19], [21], [26], [27], [29], [30], [145], [146], [147], [148], [149], [150], [151], including those leveraging *large language models* (LLMs) for code translation [29], [30], [61], [62], [63], [150]. Although prior research has shown the potential of using LLMs for code translation, there is a dearth of research on understanding and cataloging their limitations for this task. This is an important undertaking because code translation is a complex task that requires LLMs to understand code syntax (to generate syntactically correct code) and semantics (to preserve functionality during translation) simultaneously. However, research has shown that without providing adequate context to LLMs via prompt crafting, they may only serve as "next code token" predictors, without understanding the overall task [152], [153], [154], [155].

In this work, we perform a large-scale empirical study to understand the promises and limitations of LLM-based code translation, and compare them with existing non-LLM-based translation approaches. We also perform a preliminary investigation of how providing more context about incorrect translations improves the results. Our study answers the following research questions:

**RQ1: Effectiveness in Code Translation (§4.3).** (*RQ1.1*) How do state-of-the-art general and code LLMs perform in code translation? (*RQ1.2*) What are the outcomes of unsuccessful translations?

**RQ2: LLM-Based Translation Bugs (§4.4).** (*RQ2.1*) What are the different types of underlying root causes (translation bugs) for unsuccessful translations? (*RQ2.2*) How prevalent are these bugs in unsuccessful translations? (*RQ2.3*) How do translation bugs in *real-world projects* differ from those in *crafted benchmarks*?

**RQ3: Comparison with Alternative Approaches (§4.5).** How do state-of-the-art non-LLM-based techniques perform in code translation and what types of translation bugs do they introduce?

**RQ4: Mitigating Translation Bugs (§4.6**). To what extent do the proposed *prompt-crafting techniques* resolve translation bugs?

To investigate the RQs, we collected 1,700 executable code samples from *three* well-known datasets (CodeNet [130], Avatar [128], and EvalPlus [67]) and *two* open-source projects (Apache Commons CLI [65] and Python Click [66]), covering *five* PLs (C, C++, Go, Java, and Python). To perform translation, we selected *seven* LLMs: GPT-4 [32], *three* open-source LLMs from the Hugging Face Open LLM Leaderboard [156] (Llama 2 [55], TheBloke-Vicuna [157], and TheBloke-Airoboros [158]), and *three* recent code LLMs (StarCoder [34], CodeGeeX [36], and CodeGen [35]).

We performed 43,379 translations across all LLMs, measuring translation success against the tests provided with the code samples. This produced 11.94% successful translations on average (median 5.3%), with GPT-4 (47.3% success rate) and StarCoder

(14.5% success rate) being the best-performing models (details in §4.3.1). On real-world projects, the LLMs were largely ineffective, with success rates of 8.1% for GPT-4 and 0% for the rest of the models.

We also conducted a systematic study to understand the root causes of unsuccessful translations and create a taxonomy of translation bugs. The process involved eight human labelers and took 630 person-hours in total, focusing on 1,748 buggy translations by GPT-4. It was conducted in two phases: in Phase 1, a draft taxonomy was created from buggy translations for one language pair; in Phase 2, the taxonomy was used for other language pairs and extended, if needed, to label buggy translations (details in §4.4). The resulting bug taxonomy is structured into 15 categories and five groups (details in §4.4.1). Some notable findings are: (1) identifying suitable data type in the target PL that preserves the source behavior is challenging, (2) identifying equivalent APIs in the target language or implementing the API functionality can introduce bugs, and (3) replacing language-specific features, such as method overloading and annotations, can be challenging, especially in real-world projects. Another important dimension of this study is comparing LLM-based translation with existing non-LLM-based techniques, namely, CxGo [19], C2Rust [18], and JavaToCsharp [21].[1] The comparison shows that LLM- and non-LLM-based translation techniques provide different and unique advantages, suggesting that an ultimate solution for code translation should combine both techniques (details in §4.5).

Our study reveals that, often, providing only the source code may be insufficient for achieving correct code translation. To that end (and also motivated by recent research on LLM-based bug repair [68], [70]), we propose an iterative prompting approach that incorporates additional informative context in prompts corresponding to the previously unsuccessful translation, including the code, stack trace, error message from failing execution, and/or test input and expected output from failing test cases. Our results

[1]We also considered other approaches (mppSMT [26], Tree2Tree [27], and Sharpen [20]) but could not compare against them due to lack of tool/artifact availability (§4.5).

show prompt crafting increases the success rate by 5.5%, on average, across the studied LLMs, with the largest improvement, of 12%, occurring for GPT-4 (details in §4.6). Although these results are encouraging, they indicate considerable scope for improvement, likely through a combination of program analysis techniques and LLMs.

To our knowledge, we are the first to (1) provide a systematic bug taxonomy facilitating a deeper understanding of error modalities in LLM-based code translation, (2) study translation of real-world projects, (3) investigate the effectiveness of prompt crafting in mitigating translation bugs, and (4) compare non-LLM and LLM-based translation approaches. Our key contributions are:

1. **A comprehensive evaluation of LLM-based code translation.** We perform a large-scale evaluation of code translation using multiple general and code LLMs. We consider the recently released LLMs, and our evaluation includes real-world projects in addition to three crafted benchmarks.

2. **A taxonomy of translation bugs.** Our study offers the first taxonomy of bugs introduced by LLMs during code translation. We also compare the nature of these bugs in LLM and non-LLM-based approaches to understand their strengths and weaknesses.

3. **Prompt crafting to enhance code translation.** A set of heuristics for prompt crafting that provides proper contexts to LLMs to improve their effectiveness in code translation.

4. **Artifacts.** Our artifacts, including manual labeling and automation scripts for evaluating LLMs, are publicly available [40].

### 4.2 Empirical Setup

**Subject LLM Selection.** General LLMs are pre-trained on textual data, including natural language and code, and can be used for a variety of tasks. In contrast, code

Table 1: Overview of subject LLMs. TB: TheBloke.

| Modality | Code | | | Text | | | |
|---|---|---|---|---|---|---|---|
| **Models** | CodeGen | CodeGeeX | StarCoder | GPT-4 | Llama 2 | TB-Airboros | TB-Vicuna |
| **Size** | 16B | 13B | 15.5B | - | 13B | 13B | 13B |
| **Context Window** | 2048 | 2048 | 2048 | 8192 | 4096 | 2048 | 2048 |
| **Release Date** | May'23 | Mar'23 | May'23 | Mar'23 | Jul'23 | May'23 | May'23 |

LLMs are specifically pre-trained to automate code-related tasks. Due to the empirical nature of this work, we were interested in assessing the effectiveness of both LLM categories in code translation. For code LLMs, we selected the top three models released recently (in 2023), namely CodeGen [35], StarCoder [34], and CodeGeeX [36]. For general LLMs, we selected the top three models with size 20B parameters or less from the Hugging Face Open LLM Leaderboard [156].[2] The constraint on the number of parameters was imposed by our computing resources, resulting in the selection of Llama 2 [55], TheBloke-Airoboros [158], and TheBloke-Vicuna [157]. We also included GPT-4 [32] in our study. Table 1 summarizes characteristics of the selected LLMs.

**Subject PLs Selection.** We used the following criteria to select the subject PLs: (1) popularity of the language based on the TIOBE index [159], (2) inclusion of different programming paradigms, e.g., procedural, object-oriented, and functional, and (3) availability of high-quality datasets in the given PL. To make the manual effort involved in taxonomy construction manageable, we selected five PLs that met the inclusion criteria—C, C++, Go, Java, and Python.

**Dataset Collection and Pre-Processing.** To ensure the comprehensiveness of our findings and claims in understanding the nature of LLM translation bugs, we were interested in datasets used in prior studies as well as real-world projects. The former consists of small programs, likely to be less challenging for LLMs to translate, and the latter assesses the complexity of LLM translation in real-world settings.

The first six columns of Table 2 present the selected datasets and statistics about

[2]The Open LLM Leaderboard ranking is quite dynamic, and our selection is drawn from the ranking at the time of our experimentation.

Table 2: Performance of subject LLMs in translating code from different studied datasets. The best performance by general and code LLMs are highlighted in teal and violet, respectively. The final performance is computed over the average of each dataset.

| **Dataset** | **Source Language** | **Source Samples** | **#Tests** | **Target Language** | **#Translations** | **(%) Successful Translations** | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | **CodeGen** | **CodeGeeX** | **StarCoder** | **GPT-4** | **Llama 2** | **TB-Airoboros** | **TB-Vicuna** |
| CodeNet [130] | C | 200 | 200 | C++, Go, Java, Python | 800 | 23.4% | 14.9% | **42.0%** | **83.0%** | 14.9% | 18.8% | 4.4% |
| | C++ | 200 | 200 | C, Go, Java, Python | 800 | 14.0% | 3.6% | **39.1%** | **80.0%** | 9.5% | 8.3% | 3.4% |
| | Go | 200 | 200 | C, C++, Java, Python | 800 | 14.3% | 5.9% | **42.0%** | **85.5%** | 16.9% | 6.6% | 0.9% |
| | Java | 200 | 200 | C, C++, Go, Python | 800 | 21.3% | 10.3% | **30.3%** | **81.3%** | 13.9% | 6.5% | 0.1% |
| | Python | 200 | 200 | C, C++, Go, Java | 800 | 17.5% | 7.3% | **33.3%** | **79.9%** | 11.0% | 6.5% | 1.0% |
| Total/Average (CodeNet) | - | 1,000 | 1,000 | - | 4,000 | 18.1% | 8.4% | **37.3%** | **82.0%** | 13.2% | 9.3% | 2.0% |
| Avatar [128] | Java | 249 | 6,255 | C, C++, Go, Python | 996 | 8.1% | 1.8% | **11.9%** | **70.8%** | 1.8% | 5.0% | 0.0% |
| | Python | 250 | | C, C++, Go, Java | 1,000 | 3.8% | 1.6% | **14.2%** | **52.2%** | 4.7% | 0.9% | 0.9% |
| Total/Average (Avatar) | - | 499 | 6,255 | - | 1,996 | 5.9% | 1.7% | **13.0%** | **61.5%** | 3.2% | 3.0% | 0.4% |
| EvalPlus [67] | Python | 164 | 2,682 | Java | 164 | 16.5% | 3.7% | **22.0%** | **79.3%** | 1.2% | 14.0% | 7.9% |
| Commons CLI [65] | Java | 22 | 310 | Python | 22 | 0.0% | 0.0% | 0.0% | **13.6%** | 0.0% | 0.0% | 0.0% |
| Click [66] | Python | 15 | 611 | Java | 15 | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% | 0.0% |
| Total/Average (All) | - | 1,700 | 10,858 | - | 6,197 | 8.1% | 2.8% | **14.5%** | **47.3%** | 3.5% | 5.3% | 2.1% |

them (more information in the artifact website [40]). These datasets are accompanied by test cases to validate code translation. For CodeNet and Avatar, the tests are input data and expected output, while EvalPlus and real-world projects have unit tests (JUnit and pytest). For EvalPlus, we manually translated and verified the corresponding pytests to JUnit tests. The translation of real-world projects never reached test execution, as it produced syntactically incorrect code (more discussion in §4.3).

For real-world projects, we focused on Java and Python, the most popular languages among our subject PLs. Our goal was to translate reasonably complex and well-maintained software exclusively written in Java or Python. To that end, we selected projects available in both PLs providing APIs for command-line processing and selected Apache Commons CLI [65] (Java) and Click [66] (Python). To fit the source language code into the limited LLM context window, we broke them down into classes and files and removed all comments.

**Compute Resources.** To perform inference on all subject LLMs, we used 16 A100 80GB memory GPUs. For evaluating the generated translations, we used Python 3.10, g++ 11, GCC Clang 14.0, Java 11, Go 1.20, Rust 1.73, and .Net 7.0.14 for Python, C++, C, Java, Go, Rust, and C#, respectively.

Table 3: Breakdown of the unsuccessful translations produced by subject LLMs based on outcome. All values are in %.

| **Source Language** | **C** | | | | **C++** | | | | **Go** | | | | **Java** | | | | **Python** | | | | **Total** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Target language** | **C++** | **Go** | **Java** | **Python** | **C** | **Go** | **Java** | **Python** | **C** | **C++** | **Java** | **Python** | **C** | **C++** | **Go** | **Python** | **C** | **C++** | **Go** | **Java** | |
| Compilation Errors | 68.9 | 93.5 | 76.4 | 56.9 | 93.2 | 94.6 | 77.0 | 61.6 | 86.7 | 83.3 | 82.4 | 55.9 | 82.4 | 78.4 | 96.6 | 57.4 | 79.9 | 73.4 | 86.0 | 72.4 | **77.8** |
| Runtime Errors | 9.4 | 2.3 | 10.7 | 21.9 | 0.1 | 1.2 | 11.2 | 22.9 | 0.2 | 0.2 | 12.7 | 19.3 | 1.2 | 0.4 | 0.8 | 27.1 | 0.4 | 0.4 | 10.0 | 14.8 | **8.4** |
| Functional Errors | 20.5 | 3.7 | 13.0 | 20.6 | 6.7 | 4.1 | 11.7 | 15.1 | 12.9 | 16.3 | 4.7 | 24.6 | 15.8 | 19.9 | 2.5 | 15.1 | 19.0 | 24.8 | 3.9 | 12.5 | **13.4** |
| Non-terminating Execution | 1.3 | 0.4 | 0.0 | 0.5 | 0.0 | 0.2 | 0.1 | 0.4 | 0.2 | 0.2 | 0.2 | 0.2 | 0.7 | 1.3 | 0.1 | 0.3 | 0.8 | 1.3 | 0.1 | 0.3 | **0.4** |

### 4.3 LLM-Based Code Translation

We prompted each subject LLMs with 6,197 translation problems corresponding to 31 translation pairs shown in Table 2, i.e., 20 pairs from CodeNet, eight pairs from Avatar, and one pair each for EvalPlus, Commons CLI, and Click. Through RQ1, we evaluate the effectiveness of LLMs in code translation (RQ1.1) and the outcomes of incorrect translations (RQ1.2).

#### 4.3.1 Effectiveness of LLMs in Code Translation

We refer to the LLM prompting in this experiment as *vanilla prompting*, where each prompt contains four pieces of information: (1) instructions in natural language to perform the translation task, (2) source language (`$SOURCE_LANG`), (3) target language (`$TARGET_LANG`), and (4) the code to be translated (`$SOURCE_CODE`). We followed the templates similar to those we found in the artifacts, papers, or technical reports associated with each model. Figure 1 shows the three templates used for vanilla prompting of our subject LLMs. Our prompt template for CodeGeeX slightly differs from what is used in their paper [36]. Specifically, their prompt template includes imports, class declaration, and method signature of the translation [160]. However, this is an *unrealistic approach* because such ground truth does not exist and requires human involvement for each translation. Moreover, the code to be translated (from real-world projects or crafted benchmarks) often contains several methods, making it impossible to use the same template.

We consider a translation successful if it compiles, passes runtime checks, and existing tests pass on the translated code. We do not consider static evaluation metrics such as

| GPT-4 | CodeGeeX | Other models |
|---|---|---|
| $SOURCE_CODE<br>// Unformatted source code<br>Translate the above $SOURCE_LANG code to $TARGET_LANG. Print only the $TARGET_LANG code, end with comment "\|End-of-Code\|". | code translation<br>$SOURCE_LANG:<br>$SOURCE_CODE<br>// Unformatted source code<br>$TARGET_LANG:<br>// Code generated | $SOURCE_LANG:<br>$SOURCE_CODE<br>// Unformatted source code<br>Translate the above $SOURCE_LANG code to $TARGET_LANG.<br>$TARGET_LANG:<br>// Code generated |

Figure 1: Vanilla prompting templates.

exact match, syntax match, dataflow match [126], CodeBLEU [126], and CrystalBLEU [127] because our goal is to validate (compile and execute) the translations. Static metrics can also be misleading in code synthesis [129]—i.e., LLMs may achieve reasonably high numbers for these metrics, but generate code that cannot be executed due to compilation or runtime errors [128], [129]. The last seven columns of Table 2 show the detailed results of vanilla prompting of subject LLMs for code translation. We next discuss our key observations.

1. Except for GPT-4 and StarCoder, all other models performed poorly. The biggest surprise here is CodeGeeX, a model trained explicitly for code translation. We believe this result is because, as mentioned, we excluded information about the translated code (imports, class declaration, and method signature) in the prompt. Such information is typically not available and non-trivial to compute. (To check the correctness of our results, we repeated their experiments with their template and ours, which resulted in the pass@1 dropping from 25.6% to 0.02% on their dataset, HumanEval-X.)

2. There is a strong correlation between the average number of tests per translation sample and unsuccessful translation (correlation coefficient $r$ ranging from 0.64 to 0.85 for all models). That is, the more rigorous the existing test suite, the better it can evaluate if a translation preserves functionality.

3. There is no consistent pattern between unsuccessful translations and source/target language, but translating *to* Go results in more compilation errors due to its strict syntax constraints, e.g., forbidding unused variables or imports.

4. The LLMs fail to translate real-world projects. This is mainly because crafted benchmark programs are simpler, without complex dependencies or use of language features, e.g., annotations, inheritance, etc. Moreover, in a real-world setting, translating files/methods in isolation, even if successful, may fail at the project level. That said, further manual investigation showed that for the Commons CLI, three out of 22 translated files could be compiled using `py_compile` [161]. These simple classes consist of (1) an exception class with only one method, (2) an interface with two method declarations, and (3) a utility class with two simple methods.

### 4.3.2 Outcome of Unsuccessful Translations

The previous research question shows that most of the subject LLMs are yet to achieve a reasonable performance for code translation, even on crafted benchmarks, let alone real-world projects. At the next step, we were interested to understand if this is due to a lack of understanding of code syntax or semantics by LLM. To do this, we classify unsuccessful translations based on their error outcome: (1) *Compilation Error*, where translated code cannot be compiled, (2) *Runtime Error*, where translated code compiles but fails at runtime with an exception, (3) *Functional Error*, where the translated code compiles and executes successfully but results in test failure, and (4) *Non-terminating Execution*, where the translated code compiles and executes, but does not terminate (encountering an infinite loop or waiting on user input).

Figure 2 and Table 3 show the results of this experiment for each model—accumulated for all subject PLs—and for each subject PL—accumulated for all subject models, respectively. From these results, we observe that most unsuccessful

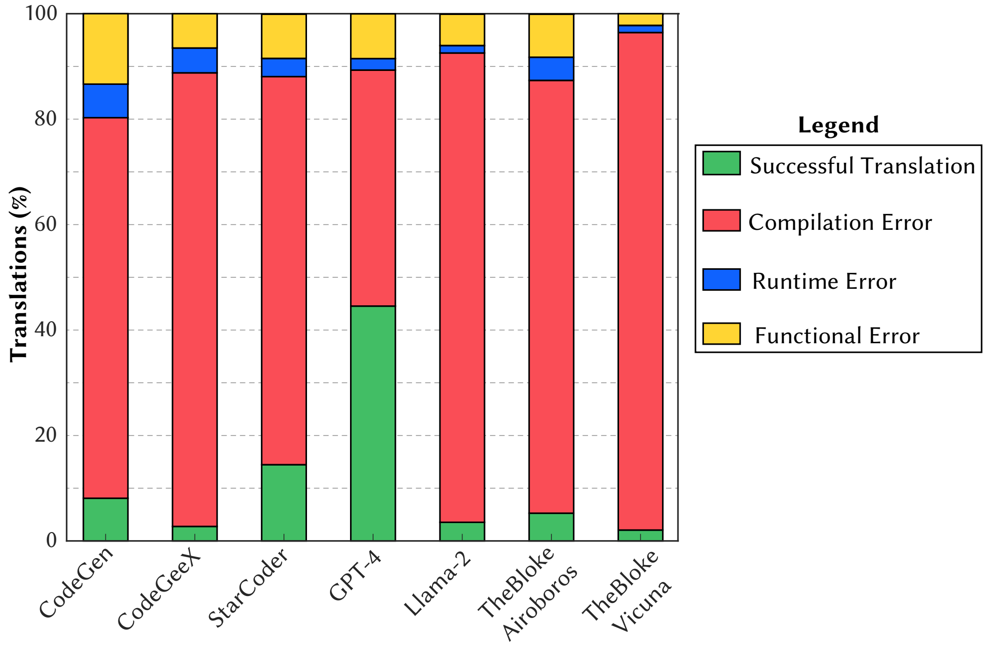



Figure 2: Outcome of code translations using subject LLMs.

translations result in compilation errors (77.8%), meaning both general and code LLMs have difficulty understanding code syntax. Further breakdown of the results per PLs shows that Go and C have comparatively stricter syntax, while it is easier for LLMs to generate syntactically correct Python code. [3] The next most common effect of unsuccessful translation is a functional error (13.4%), demonstrating that often translated code does not preserve the behavior of the source program.

### 4.4 LLM-Based Translation Bugs

To understand the nature of translation bugs, we performed a deep analysis by manually investigating the root cause of unsuccessful translations. Through the following three research questions, we introduce our comprehensive taxonomy of translation bugs (RQ2.1), investigate the prevalence and distribution of each bug category across unsuccessful translations (RQ2.2), and discuss the peculiar characteristics of bugs in real-world projects (RQ2.3).

[3] We used `py_compile` [161] to check syntax-related bugs in Python.

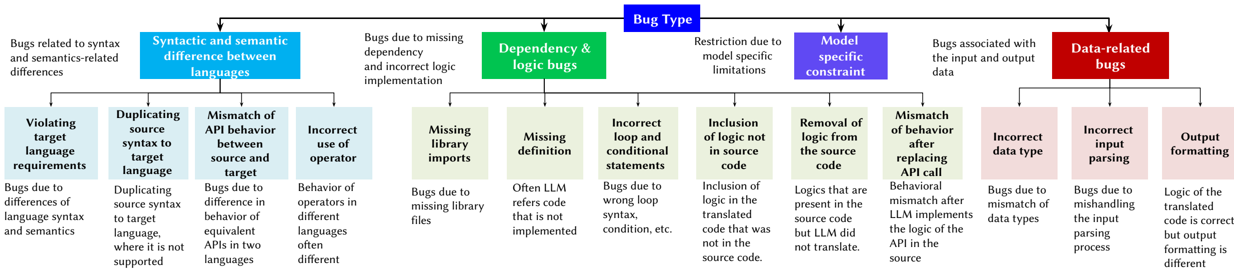



Figure 3: Taxonomy of bugs introduced while translating code using LLM. The "other" category/group is not shown here.

### 4.4.1 Taxonomy of Translation Bugs

Our initial investigation showed that GPT-4 is the best-performing model and compared to others, its translations exhibit a variety of quality bugs worth investigating. To manage the manual effort for understanding and labeling bugs for building the taxonomy, we focused on 1,748 unsuccessful translations from GPT-4.

**Methodology.** The manual construction of the taxonomy involved eight human labelers, who are researchers or software engineers in the industry, and involved unsuccessful translations from all 31 translation pairs in Table 2. We built the taxonomy in two phases. In the first phase, we used the CodeNet Java-to-Python translations. Each labeler created a taxonomy independently by examining the unsuccessful translations. Then, we combined the individual taxonomies to create a consolidated taxonomy, which served as the initial taxonomy for phase 2. In the second phase, each of the remaining 30 translation pairs was examined by two labelers to assign bug categories to unsuccessful translations. Whenever a new category came up (i.e., a bug could not be covered by the existing categories in the taxonomy), the entire team met to discuss the new category, add it to the taxonomy, and re-label the affected bugs. After completing their labeling tasks, the two labelers assigned to a translation pair met to discuss their labeling, resolve discrepancies, and create the final labeling for the translation pair. The entire exercise took about 630 person-hours and produced a taxonomy with 15 bug categories organized into five groups ("model specific constraints" group does not have any sub-category). In the rest of this section, we discuss the taxonomy groups and bug categories together with illustrative examples.

**Syntactic and semantic differences between the source and the target languages.** There are five bug categories in this group that relate to the failure of an LLM in appropriately handling the syntactic or semantic differences between PLs.

*A1: Violating target language requirements.* Each PL has its own set of rules that the code must adhere to. For example, in Java, any executable code must be wrapped in a method within a class, whereas in Python, this is not a requirement. Similarly, unused imports in Go result in compilation errors, but not in other PLs.

*A2: Duplicating source syntax to the target language.* LLMs often copy the source PL syntax even if they are not available in the target PL. In an unsuccessful translation below from C++ to Go, the subject LLM does not replace `atan2l` API (which is specific to C++ and does not exist in Go) with an appropriate equivalent.

```
const ld PI = atan2l(0, -1); # Original C++ code
PI = atan2l(0, -1)           # Incorrect Go code
```

*A3: Mismatch of API behaviors in the source and target.* Library APIs are frequently used in programs in any PL. During translation, API calls in the source need to be either mapped to equivalent API calls in the target PL or implemented from scratch. In the former case, we noted that LLMs often map source APIs incorrectly to the target PL. The following code fragment illustrates such an example where the Java `String.substring()` API is incorrectly mapped to the Go `strings.IndexByte()` API method.

```
S.substring(i, i + 1)   # Original Java code (returns String)
strings.IndexByte(S, i) # Incorrect Go code (returns Int64)
```

*A4: Incorrect use of operators.* The supported operators and their syntax can vary among PLs. For example, `//` in Python represents floor division, for which Java has no corresponding operator. To achieve that behavior in Java, a division must be followed by a call to `Math.floor()`. LLMs can fail in translating such cases.

```
i = i // 10;  # Original Python code (floor division)
i = i / 10;  # Incorrect Java code (division)
```

**Dependency and logic bugs in the translated code.** These bugs pertain to incorrect dependencies and logic of translated code.

*B1: Missing library imports.* Import statements are used to load libraries and/or application classes/modules used in code. In several unsuccessful translations, we found that the translated code had missing or incorrect imports.

*B2: Missing definition.* LLMs can omit definitions/implementation of data types, methods, etc. In the unsuccessful translation below, the original C++ code, `main()` calls method `solve()` that is implemented in the same source file. However, after translation to Go, although the call to `solve()` remains, its definition is removed.

```
void solve(){...} # Original C++ code
signed main(){...solve();} # Incorrect Go code
```

*B3: Incorrect loop and conditional statements.* This category covers bugs in translating loops and conditional statements. The following code illustrates the incorrect translation of a Java loop to Python array slicing, resulting in an off-by-one error in the translated code, where the sum excludes the value of `x[200010-k-1]`.

```
for(int i = 0; i <= 200010 - k - 1; i++) ans += x[i]; # Java code
ans = sum(x[:200010 - k - 1]) # Incorrect Python code
```

*B4: Inclusion of logic not in source code.* LLMs can generate code that is unrelated to any logic in the source program, thereby, causing the translated code to diverge from the source behavior. In the following example, `max` is initialized to `-1` in the source. However, after translation, it is initialized to a different value; the LLM thus adds logic that does not exist in the source program.

```
int max = -1; # Original C code
max_h = max(h) # Incorrect Python code
```

*B5: Removal of logic in the source code.* LLMs often fail to translate part of the source program correctly. For example, in several cases lines such as `#define MAX 101` in C are removed.

*B6: Mismatch of behavior after replacing API call.* These bugs occur when a source API call is translated to custom logic instead of an equivalent API in the target PL (A3 bugs).

```
return Collections.unmodifiableList(requiredOpts); # Java code
return self.required_opts # Incorrect Python code
```

An erroneous translation from Commons CLI is demonstrated in the example above, where Java's `Collections.unmodifiableList()`, which typically provides an immutable list is converted to a mutable list in Python.

**Data-related Bugs.** Data plays an integral role in the code. We found various bugs caused by wrong assumptions about the input data, mismatch of data types, etc.

*C1: Incorrect data type.* This category of bugs pertains to incorrect types assigned to variables. The following example, taken from the translation of Commons CLI to Python, illustrates an incorrect type assignment to a field.

```
public static Class<File[]> FILES_VALUE=File[].class; # Java code
FILES_VALUE = List[os.path] # Incorrect Python code
```

*C2: Incorrect input parsing.* Programs that read data from the input stream (i.e., `stdin` or other sources expect the data to be in a specific format. Such programs are often translated incorrectly, as illustrated by the following Python-to-C translation. The second input line contains three integer values all of which are read into an array in the Python code (line 6), whereas the C code reads only two values and assigns 0 for the third value (lines 2–4). This causes the wrong result to be computed in line 6.

```
1 --------------- INPUT ---------------
2 2
3 3 5 2
4 4 5
5 ------------ SOURCE CODE ------------
6 N = int(input())
7 A = [.. input().split()]
8 ...
9     d = min(A[i+1], B[i])
```

```
1 ---------- TRANSLATED CODE ----------
2 int A[N+1], B[N], N;
3 for(int i=0; i<N; i++) {
4     scanf("\% A[N] = 0;
5 ...
6     d = A[i+1] < B[i] ? \
7     A[i+1] :
8     B[i];
9 ...
```

**C3: Output formatting.** We found that often, even if the translated code logic is correct, the output is formatted differently. In the following example, the source Java code prints ‘H’ followed immediately by an integer value, whereas the translated Python code prints a space between ‘H’ and the integer value.

```
System.out.print("H"); System.out.println(Y - 1988); # Java code
print("H", Y - 1988) # Incorrect Python code
```

*D: Model-Specific Bugs.* Some bugs are specific to the design of the LLMs used. For instance, having natural language in-between the code, exceeding token size, etc., causing compilation errors or no output to be generated. We also observed a group of unsuccessful translations—which we refer to as *E: Others*—related to our experiment setup (e.g., memory issues). Given that they do not represent LLM-introduced bugs, we do not include them in the taxonomy.

Table 4: Types of bugs introduced during code translation by GPT-4 and their occurrences. This table includes all the subject datasets including the real-world projects. All values are in %.

| Source Language | C | | | | C++ | | | | Go | | | | Java | | | | Python | | | | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Target Language | C++ | Go | Java | Py | C | Go | Java | Py | C | C++ | Java | Py | C | C++ | Go | Py | C | C++ | Go | Java | |
| A1: Violating target language requirements | 23.1 | 78.2 | 57.7 | 20.0 | 25.9 | 58.5 | 23.5 | 11.3 | 8.3 | 37.5 | 3.6 | 10.0 | 20.4 | 2.7 | 55.8 | 4.6 | 14.5 | 5.8 | 41.4 | 11.2 | 24.3 |
| A2: Duplicating source syntax to the target language | 0.0 | 0.0 | 0.0 | 0.0 | 18.5 | 3.1 | 5.9 | 0.0 | 2.8 | 4.2 | 3.6 | 2.5 | 0.9 | 6.8 | 0.6 | 1.1 | 4.6 | 1.2 | 1.8 | 2.9 | 2.4 |
| A3: Mismatch of API behaviors in the source and target | 0.0 | 1.8 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 11.1 | 0.0 | 0.0 | 2.5 | 4.4 | 5.5 | 3.5 | 16.1 | 0.0 | 2.9 | 0.5 | 1.0 | 3.3 |
| A4: Incorrect use of operator | 0.0 | 0.0 | 0.0 | 2.2 | 0.0 | 0.0 | 2.9 | 0.0 | 0.0 | 0.0 | 3.6 | 0.0 | 0.0 | 1.4 | 0.0 | 0.0 | 1.2 | 0.0 | 0.0 | 2.4 | 0.6 |
| A: Syntactic and semantic differences between languages | 23.1 | 80.0 | 57.7 | 22.2 | 44.4 | 61.5 | 32.4 | 11.3 | 22.2 | 41.7 | 10.7 | 15.0 | 25.7 | 16.4 | 59.9 | 21.8 | 20.2 | 9.9 | 43.6 | 17.6 | 30.5 |
| B1: Missing library imports | 0.0 | 0.0 | 7.7 | 0.0 | 0.0 | 1.5 | 0.0 | 0.0 | 8.3 | 0.0 | 0.0 | 0.0 | 3.5 | 5.5 | 1.2 | 8.6 | 7.5 | 32.0 | 2.7 | 22.4 | 8.6 |
| B2: Missing definition | 0.0 | 0.0 | 7.7 | 0.0 | 11.1 | 3.1 | 14.7 | 1.9 | 13.9 | 4.2 | 0.0 | 0.0 | 2.7 | 4.1 | 0.0 | 0.0 | 3.5 | 5.8 | 0.5 | 0.0 | 2.4 |
| B3: Incorrect loop and conditional statements | 15.4 | 1.8 | 3.8 | 2.2 | 7.4 | 4.6 | 5.9 | 3.8 | 0.0 | 8.3 | 0.0 | 0.0 | 1.8 | 0.0 | 1.7 | 4.6 | 5.2 | 2.3 | 1.8 | 0.0 | 2.6 |
| B4: Inclusion of logic not in the source | 0.0 | 1.8 | 3.8 | 4.4 | 3.7 | 3.1 | 0.0 | 1.9 | 13.9 | 12.5 | 3.6 | 5.0 | 5.3 | 1.4 | 0.0 | 2.3 | 5.8 | 4.1 | 0.9 | 4.9 | 3.4 |
| B5: Removal of logic from source | 0.0 | 0.0 | 3.8 | 8.9 | 3.7 | 3.1 | 11.8 | 13.2 | 5.6 | 0.0 | 3.6 | 2.5 | 8.8 | 2.7 | 1.2 | 2.9 | 2.3 | 1.7 | 0.9 | 3.4 | 3.3 |
| B6: Mismatch of behavior after replacing API call | 0.0 | 0.0 | 0.0 | 0.0 | 3.7 | 1.5 | 0.0 | 1.9 | 0.0 | 4.2 | 21.4 | 2.5 | 9.7 | 9.6 | 2.9 | 0.6 | 5.2 | 2.3 | 5.0 | 3.9 | 3.8 |
| B: Dependency & logic bugs in the translated code | 15.4 | 3.6 | 26.9 | 15.6 | 29.6 | 16.9 | 32.4 | 22.6 | 41.7 | 29.2 | 28.6 | 10.0 | 31.9 | 23.3 | 7.0 | 19.0 | 29.5 | 48.3 | 11.8 | 34.6 | 24.2 |
| C1: Incorrect data type | 7.7 | 1.8 | 0.0 | 11.1 | 7.4 | 15.4 | 26.5 | 1.9 | 27.8 | 4.2 | 7.1 | 7.5 | 8.8 | 8.2 | 7.6 | 0.6 | 21.4 | 15.7 | 8.2 | 21.5 | 11.5 |
| C2: Incorrect input parsing | 23.1 | 7.3 | 11.5 | 40.0 | 7.4 | 4.6 | 0.0 | 60.4 | 5.6 | 0.0 | 14.3 | 60.0 | 9.7 | 19.2 | 11.6 | 32.2 | 11.0 | 15.7 | 24.1 | 10.2 | 18.1 |
| C3: Output formatting | 23.1 | 0.0 | 0.0 | 4.4 | 0.0 | 0.0 | 0.0 | 1.9 | 0.0 | 0.0 | 7.1 | 0.0 | 1.8 | 9.6 | 3.5 | 5.2 | 10.4 | 1.7 | 5.0 | 2.4 | 3.9 |
| C: Data-related Bugs | 53.8 | 9.1 | 11.5 | 55.6 | 14.8 | 20.0 | 26.5 | 64.2 | 33.3 | 4.2 | 28.6 | 67.5 | 20.4 | 37.0 | 22.7 | 37.9 | 42.8 | 33.1 | 37.3 | 34.1 | 33.5 |
| D: Model specific constraints | 7.7 | 7.3 | 3.8 | 4.4 | 11.1 | 1.5 | 8.8 | 1.9 | 0.0 | 0.0 | 25.0 | 0.0 | 8.0 | 4.1 | 9.9 | 14.4 | 2.9 | 2.3 | 5.9 | 7.8 | 6.6 |
| E: Others | 0.0 | 0.0 | 0.0 | 2.2 | 0.0 | 0.0 | 0.0 | 0.0 | 2.8 | 25.0 | 7.1 | 7.5 | 14.2 | 19.2 | 0.6 | 6.9 | 4.6 | 6.4 | 1.4 | 5.9 | 5.1 |

### 4.4.2 Prevalence of LLM-based Translation Bugs

We now present the results for RQ2.2, showing the prevalence of bugs in different categories of our taxonomy. Among the 6197 attempted translations over 31 language pairs (Table 2), there were 1558 translation failures. We manually checked and labeled these failures to identify 1748 bugs. In many cases, an unsuccessful translation has multiple bugs that belong to different categories; in these cases, the translation gets multiple labels. Table 4 presents detailed results on the prevalence of translation bugs. In this section, we delve deeper into the characteristics and prevalence of these bugs.

**Finding 1:** More than one-third (33.5%) of the translation bugs are data-related bugs.

As the data for bug group C in Table 4 show, a large proportion of the LLM-introduced bugs is related to data types, parsing input data, and output formatting issues, together accounting for 33.5% of all bugs. These bugs are particularly prevalent for Python to C, C++, and Go translations, constituting over 43%, 33%, and 37% of the bugs, respectively, in those translations. Within data-related bugs, our manual investigation found several unique patterns.

**Finding 1a:** Among the data-related bugs, most (54% of the data-related bugs and 18.1% of the total bugs) are due to incorrect parsing of inputs.

As discussed in *data-related bugs*, programs that take external inputs contain input-parsing logic, assuming the data to adhere to certain formats, and LLMs often make mistakes while translating this logic; the C2 bug category example in *data-related bugs* illustrates this. A major reason for the prevalence of this bug is that two of our datasets (CodeNet and Avatar) consist of programs that read data from the input stream. For EvalPlus and the real-world projects, the occurrence of this bug is lesser (one for EvalPlus and none for the real-world projects).

**Finding 1b:** Choosing the correct data type in the target PL is a crucial step that accounts for 34.3% of all data-related bugs and 11.5% of all bugs.

Assignment of correct data types in the translated code (11.5% of translation bug) is a major challenge. These bugs occur due to incorrect choice of the data type, differences between behaviors of equivalent data types across PLs, and differences in type systems of the source and target PLs. The example for the C1 bug category in *data-related bugs* shows an instance of wrong choice of data type in the target PL, where the Java type `Class<File[]>` is converted to list of path objects in Python. To illustrate an example of equivalent types with different behaviors in source and target PLs, consider the code fragments shown below, where the function `mean_a_d()` in the Python code (left) takes a list of `float` as input. The test case for the function (line 2) uses a large number as test data. The translated code, shown on the right, looks correct and maps Python `float` to the Java `Float` type. However, the equivalent Java test case (line 2 on right) fails because of a fundamental difference between Python `float` and Java `Float`: the former uses 64-bit precision whereas the latter uses 32-bit precision. The Java code thus cannot handle the large test data value, which works fine in Python.

```
------------ SOURCE CODE ------------
def mean_a_d(numbers: List[float])
    -> float:
    self.assertEqual(0.0,
    mean_a_d([1e+308]))
    ...
```

```
---------- TRANSLATED CODE ----------
public static float
    meanAD(List<Float> n) ...
void testCode() {assertEquals(0.0,
    meanAD(Arrays.asList(1e+308f)));
}
```

Incorrect data type bugs can also occur due to differences between the type systems of the source and target PLs. For instance, while translation code in a dynamically typed PL (e.g., Python) to a statically typed PL (e.g., Java), preserving the behavior of source types can be challenging.

**Finding 2:** A significant proportion, 30.5%, of the translation bugs occur due to syntactic and semantic differences between the source and target PLs; almost 80% of these (24.3% of all bugs) are caused by violation of target language requirements.

Almost one-third of the bugs occur due to reasons, such as violating target language requirements, duplicating source syntax to the target PL, behavioral differences of APIs and operators, etc. *Syntactic and semantic differences between the source and the target languages* illustrates several bugs in this group. For instance, the following translated code violates syntactic constraints of the target language: the variable named `ll` in the Python code (left) is translated verbatim to C++, which results in a compilation error as `ll` is a reserved keyword in C++ (used for long-long data).

```
------------ SOURCE CODE ------------
ll = - 10 ** 18 - 1
```

```
---------- TRANSLATED CODE ----------
ll ll = -1e18 - 1;
```

Another example of such a bug is the declaration order of methods, where some languages (e.g., Go) permit methods to call another declared subsequently, whereas others (e.g., C++) restrict calls to previously declared methods only. Thus, maintaining the wrong declaration order of methods could result in compilation errors.

**Finding 2a:** Replacing an API call with another API call in the target PL can result in bugs.

As illustrated for the A3 bug category in *syntactic and semantic differences between the source and the target languages*, LLMs can incorrectly map source APIs to the APIs available in the target language. Table 4 shows that 3.3% of all bugs fall in this category. The following example illustrates an API-mismatch bug, where the LLM replaces the Python `accumulate()` API with `IntStream.concat()` in Java, which can be used in an equivalent manner. However, the LLM erroneously adds `reduce(count).getAsInt()` to reduce the result to an integer value instead of converting the return value of `IntStream.concat()`, an integer stream, to a list/array of integers. This results in an incorrect return value in the two programs: a list of integers in Python and an integer in Java.

```
list(accumulate([0] + list(range(1, n)), count)) # Python code
int[] cumsum = IntStream.concat(IntStream.of(0), \
IntStream.range(1, n)).reduce(count).getAsInt(); # Incorrect Java code
```

**Finding 3:** 24.2% of the translation bugs are related to incorrect code logic and missing dependencies in the target PL, with missing imports being the dominant category.

LLMs can often translate the source logic incorrectly or miss dependencies. Missing imports is the most frequently occurring bug category (35.5% of the bugs in the group and 8.6% of the bugs overall). The other five categories in this group occur in roughly comparable numbers, ranging from 2.4% to 3.8% of the overall bugs.

### 4.4.3 Translation Bugs in Real-world Projects

Many of the bugs seen in the crafted benchmarks translation also occurred while translating Commons CLI and Click. Key among these are: removal of logic in the source (e.g., source methods and field initialization not translated), inclusion of logic not in source (e.g., implementation added for a stubbed method), missing imports, mismatch in API behaviors after translation, mismatch of behaviors after replacing API calls and incorrect data types.

However, real-world applications have much more complex code than crafted benchmarks, making them much harder for LLMs to translate. We found nine instances (all from Click) where the translated files contained natural-language text explaining the translation of the source file is infeasible or a non-trivial task for GPT-4. Some others included partial translations of methods in the class, leaving the rest untranslated (translation bug B5). This shows that LLMs, even with longer context windows, cannot capture dependencies between the methods implementing the class logic.

**Finding 4:** Real-world applications pose complex challenges for code translation, such as handling method overloading, inheritance, and code annotations, not seen in crafted datasets.

We found examples of language features used in real-world applications that LLMs can struggle with translating.

```
--------------------------------- SOURCE CODE ---------------------------------
public Options addOption(final Option opt) { ... }
public Options addOption(final String opt, final boolean hasArg,
    final String description) { ... }
```

```
------------------------------- TRANSLATED CODE -------------------------------
def add_option(self, opt):...
```

```
def add_option_arg(self, opt, has_arg, description):...
```

For instance, Commons CLI uses method overloading frequently. GPT-4 translates these to Python in different ways, some of which are correct translations, whereas others are erroneous. For an example of the latter, there are cases where GPT-4 translates overloaded Java methods by renaming them in Python to avoid overloaded method names, leaving open the work of suitably renaming all call sites to the methods to preserve the call relations.

In another instance, overloaded methods are translated to methods with the same name, which results in broken functionality because only the last method is available as per the Python semantics, with the previous method definitions overridden.

```
------------ SOURCE CODE ------------
public static OptionBuilder
    hasArg() { }
public static OptionBuilder
    hasArg(final boolean hasArg) { }
```

```
---------- TRANSLATED CODE ----------
@staticmethod
def has_arg(): ...
@staticmethod
def has_arg(has_arg): ...
```

Commons CLI and Click also illustrate the challenges posed by the use of decorators (for adding new functionality to an existing object without modifying its structure) and annotations (for adding metadata to code). For example, Click uses the `@contextmanager` decorator on a method to wrap it with a resource manager. This needs to be translated appropriately in Java to ensure automatic resource release. We also observed cases of broken inheritance relations (resulting in missing behaviors/states) in translated code and incorrect translation of exception handling.

**Finding 5:** The effectiveness of code translation can vary considerably based on the characteristics of the source and target PLs, such as the type system, available programming APIs, metaprogramming support via decorators or annotations, etc.

There is a clear pattern of GPT-4 performing much better in translating Commons

CLI to Python than Click to Java. This is evident from not only the occurrences of successful translations (three for Commons CLI vs. none for Click) and degenerate code-generation instances (nine for Click vs none for Commons CLI), but also the translation bugs observed. This could be attributed to project-specific complexity characteristics (e.g., the largest source file has $2,436$ NCLOC in Click and 358 NCLOC in Commons CLI), and it is hard to generalize from limited observations, but language features and the available API ecosystem for a PL can have a considerable impact on the success of code translations. For example, Python-to-Java translations can be more error-prone than Java-to-Python translations, in general, because going from a dynamic type system to a static one can be harder for an LLM to reason about.

> **Finding 6:** Although the translation bug categories remain the same, occurrences of bug instances and their distribution vary between crafted benchmarks and real-world projects.

The bug categories in our taxonomy are broad enough to represent translation bugs in real-world projects. However, their frequencies considerably differ. The most prevalent bugs in real-world projects are model-specific constraints (29.4%), missing imports (27.4%), and the removal of logic from the source code (15.7%), whereas for crafted benchmarks, data, syntactic, and semantic differences are more common. Moreover, the nature of these bugs is also different. For instance, in most cases, LLM's context window is not large enough to fit both project files and translated code, resulting in model-specific constraint bugs. Second, LLMs lack specific application knowledge, such as the project structure, external dependencies, and declarations outside the translation scope, i.e., the translated file. The lack of a holistic view of the application can cause bugs such as missing imports and removal of source code logic. Moreover, this limitation can cause inconsistencies in the translated code, e.g., as illustrated for Finding 4, renaming overloaded methods requires all the related call sites to be updated.

Table 5: Successful translation (in %) of non-LLM and LLM approaches. The top-2 tools are highlighted in teal and violet. **{S, T}L:** Source, Target Language. Data are in %. **TB-{A, V}**: TB-Airoboros, Vicuna.

| Dataset | SL | TL | CxGo | C2Rust | Java2C# | CodeGen | CodeGeeX | StarCoder | GPT-4 | Llama 2 | TB-A | TB-V |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **CodeNet** | C | Rust | - | **95.0** | - | 0.0 | 0.0 | 10.5 | **61.0** | 0.5 | 0.5 | 0.0 |
| | | Go | **62.3** | - | - | 33.0 | 0.0 | 12.5 | **72.5** | 5.0 | 5.5 | 11.0 |
| | Java | C# | - | - | 0.0 | 0.5 | 1.0 | **26.5** | **49.0** | 1.5 | 1.0 | 4.0 |
| **Avatar** | Java | C# | - | - | 0.0 | 0.0 | 0.0 | **10.4** | **59.2** | 2.0 | 0.8 | 0.4 |

### 4.5 LLM- vs. Non-LLM-based Translation

This section compares LLM- and non-LLM-based code translation techniques concerning their effectiveness and the differences in translation bugs. Non-LLM-based techniques are either (1) transpilers (i.e., source-to-source compilers) or cross-lingual code executors for translation (e.g., C2Rust [18], CxGo [19], Sharpen [20], and JavaToCSharp [21]), or (2) learning-based, which leverage neural machine translation techniques to convert a code from one programming language to another (e.g., mppSMT [26] and Tree2Tree [27]).

Among the mentioned tools, we used CxGo [19] and C2Rust [18] to translate C code in our CodeNet dataset to Go and Rust, and JavaToCSharp [21] to translate Java code in CodeNet and Avatar datasets to C#. Accordingly, we used our seven subject LLMs to translate code for all three PL pairs. Table 5 illustrates the success rate of both LLM and non-LLM approaches.

**Java to C#.** We observed that none of the translated code using JavaToCSharp can compile and, in fact, require heavy rewriting of library APIs (e.g., System.in and java.util.Scanner). Developers of JavaToCSharp confirmed that this tool is not built to perform full code migration. On the contrary, GPT-4 and StarCoder achieve 54.1% and 18.45% success rate in translating Java code to C#.

**Finding 7:** For C to Go, the best-performing LLM, i.e., GPT-4, achieves 10% higher success rate than non-LLM-based approach, whereas, for C to Rust, the non-LLM-based approach translates 95% of the code (34% better than best-performing LLM).

**C to Go.** CxGo transforms the C code into a common abstract syntax tree (AST) that represents both C and Go constructs. It then converts the AST into Go for translation. When compared to LLMs, we found that CxGo outperforms all the open-source models with an accuracy of 62.25%, but falls 10% behind GPT-4. The biggest caveat is that executing code generated by CxGo requires specific libraries, i.e., `stdio` for I/O operations instead of a more generic `fmt` library. In terms of the effect of the incorrect translation, 46.7% and 53.3% are due to compilation and runtime errors. Whereas, for the best-performing LLM, i.e., GPT-4, 76.3%, 12.7%, and 9.1% errors are related to compilation, test, and runtime failures.

With a further investigation on the type of translation bugs, we found that most GPT-4 translation bugs (71.8%) are related to unused imports, variables, and other target language-specific violations, followed by incorrect input parsing (10.3%). Whereas, with CxGo, the majority of the bugs are related to input parsing (26.7%) and mismatch of behavior after replacing API call (26.7%). In most cases, the input parsing related API, i.e., **`scanf`**, has been replaced by an API call that uses a custom library written specifically for the tool. Other bugs include missing definitions of API functions that are called from the translated code. The nature of these bugs is significantly different as the non-LLM-based approaches tend to have more bugs related to the custom library, while LLM-based approaches have more syntax and semantics-related bugs.

**C to Rust.** C2Rust is a transpiler that parses C code into C AST, converts the C AST into equivalent Rust AST by considering the differences in syntax, memory management, and ownership semantics between the two languages, and converts the Rust AST into Rust code. Code translation using C2Rust achieves a significantly higher

```
fn main() {
    let mut x = String::new();
    let mut y = String::new();
    io::stdin().read_line(&mut x).expect("Failed to read line");
    io::stdin().read_line(&mut y).expect("Failed to read line");
    let x: i32 = x.trim().parse().expect("Please type a number!");
    let y: i32 = y.trim().parse().expect("Please type a number!");
    println!("{} {}", x*y, 2*x + 2*y); }
```

**(a) C Code Translated to Rust by GPT-4**

```
#![allow(dead_code, mutable_transmutes, unused_assignments, unused_mut)]
extern "C" {
        fn printf(_: *const libc::c_char, _: ...) -> libc::c_int;
        fn scanf(_: *const libc::c_char, _: ...) -> libc::c_int; }
unsafe fn main_0() -> libc::c_int {
        let mut x: libc::c_int = 0;
        let mut y: libc::c_int = 0;
        x = 0 as libc::c_int;
        y = 0 as libc::c_int;
        scanf(b"%d\0" as *const u8 as *const libc::c_char,
              &mut x as *mut libc::c_int);
        scanf(b"%d\0" as *const u8 as *const libc::c_char,
              &mut y as *mut libc::c_int);
        printf(b"%d %d\n\0" as *const u8 as *const libc::c_char,
               x * y, 2 as libc::c_int * x + 2 as libc::c_int * y,);
        return 0 as libc::c_int; }
```

**(b) C Code Transpiled with C2Rust with annotations on safety issues.**

Figure 4: Comparative analysis of C-to-Rust code translation.

success rate (95%) when compared to GPT-4 (61%). Using GPT-4, 50% of the translation bugs were due to compilation errors, 39.3% were caused by runtime errors, and 10.7% were due to test failures. As for C2Rust, the 5% unsuccessful translations were due to compilation errors, the majority of them caused by unused imports.

> **Finding 8:** C2Rust generates non-idiomatic and unsafe code, whereas GPT-4 tends to generate safer and idiomatic code.

A close inspection of the translated Rust code offers several noteworthy observations. Figure 4 shows an example. For the same C code, the translation by GPT-4 (Figure 4-(a)) adheres to idiomatic Rust translations, whereas translations produced by C2Rust (Figure 4-(b)) contain unsafe code that directs the Rust compiler to bypass any safety checks. In fact, all of the C2Rust generated code are unsafe, whereas, only three GPT-4-translated code are unsafe.

- *Safety risks from compiler directives.* The use of compiler directives (see $\boxed{1}$ in Figure 4-(b)) can introduce various safety risks: (1) permitting dead code, (2) the `mutable_transmutes` directive can cause undefined behaviors and mask logical flaws, and (3) allowing unnecessary mutability can cause race conditions.

- *Use of `extern "C"` for foreign function interface.* Potential safety risks may result due to reliance on external C (see $\boxed{2}$ in Figure 4-(b)) functions such as `printf` and `scanf` without Rust's safety guarantees. Specifically, the use of `scanf` (as in $\boxed{5}$) without buffer size specifications can lead to vulnerabilities such as buffer overruns.

- *Unrestricted use of unsafe block.* The `unsafe` block in `main` function (see $\boxed{3}$ in Figure 4-(b)) circumvents Rust's safety checks for the code encompassed in that method. This increases the risk of memory safety violations. This issue is further exacerbated by the use of implicit casting (e.g., `0 as libc::c_int` as in $\boxed{4}$) in the method body, which can cause unexpected behavior.

### 4.6 Mitigating Translation Bugs

In this section, we discuss how context information pertaining to unsuccessful translations can help fix buggy translations.

***Prompt Crafting.***

Inspired by other works on prompt crafting for fixing bugs [68], [162] and how human developers would address a translation error, we propose an *iterative* prompting approach. Our hypothesis is that providing more context information to LLMs can help generate better code. Based on this hypothesis, we include the following contextual information in the revised prompts (Figure 5).

❶ **Source code and original prompt.** Here, we include the original code and the previous prompt used for translating the original code to remind LLMs about the previous task.

❷ **Incorrect translation and error details.** Here, we provide the incorrectly translated code (`$INCORRECT_TRANSLATION`), and details regarding the outcome of the translation. If it is runtime error, we provide stack trace (`$STACK_TRACE`); for compilation error, we provide error log (`$ERROR_LOG`); for test failures, we provide the incorrect output (`$INCORRECT_OUTPUT`); finally, for non-terminating execution, we provide a custom message "The program enters infinite loop."

❸ **Instructions for translation.** Here, we ask LLM to mitigate the bug and avoid including natural-language text in the response.

❹ **Expected behavior.** This part is used optionally if the prior translation was a functional error. Here, we provide test input and expected output pair for the previously wrong output.

❺ **Model-specific keyword.** This part is specific to code LLMs and contains the name of the target PL following code LLM templates.

All the prompts that we used are in the replication package [40].

***Iterative Translation Bug Mitigation.*** Our prompt-crafting technique is *iterative*. At each iteration $iter_i$, we update the prompt template (Figure 5) with information corresponding to the previously failed translation. We refer to the outcome of $iter_i$ translation as *translation patch*. At the end of each iteration, we verify if the patch results in a successful translation. If not, we utilize the outcome of the patch and build the prompt for the next iteration. The iterative mitigation can continue for a fixed number of iterations or until the percentage of successful translations at the previous iteration is smaller than a pre-defined threshold.

To evaluate the effectiveness of our iterative prompt crafting, we attempted to mitigate unsuccessful translations from RQ1 for four subject LLMs: CodeGen,

```
You were asked to translate the following
$SOURCE_LANG code to $TARGET_LANG:                          1
$SOURCE_CODE
--------------------------------------------------------------
Your response was the following $TARGET_LANG
code:
$INCORRECT_TRANSLATION                                      2
Executing your generated code gives the follow-
ing output:
${STACK_TRACE | ERROR_LOG | INCORRECT_OUTPUT}
--------------------------------------------------------------
Can you re-generate your response and translate
the above $SOURCE_LANG code to $TARGET_LANG. Do             3
not add any natural language description in your
response.
--------------------------------------------------------------
Your generated $TARGET_LANG code should take the
following input and generate the expected
output:
Input:                                                      4
$TEST_INPUT
Expected Output:
$TEST_OUTPUT
--------------------------------------------------------------
$TARGET_LANG Code: // Generated code                        5
```

Figure 5: Prompt crafting template for LLMs with the context corresponding to the outcomes of unsuccessful translation.

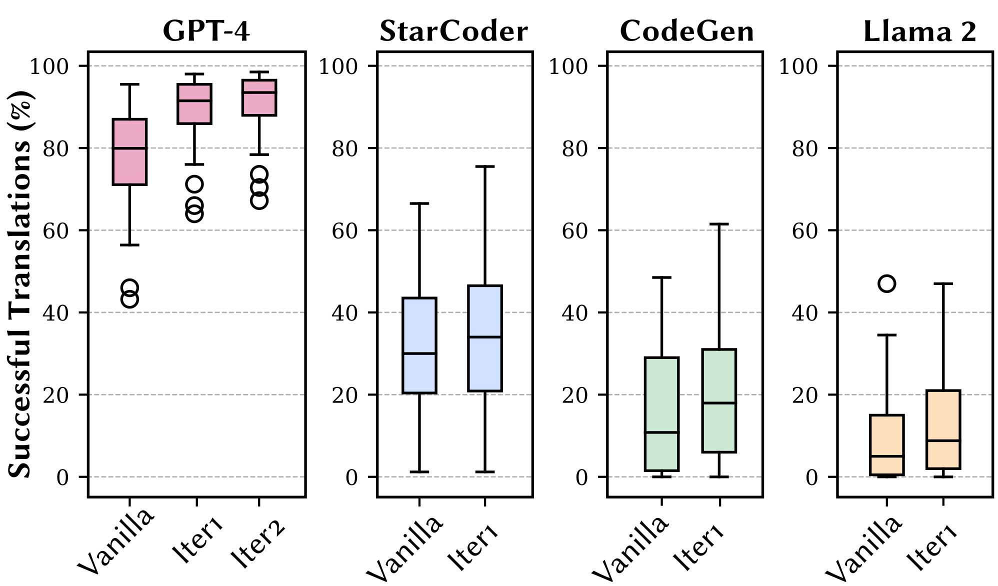



Figure 6: Effectiveness of providing error-related information in the prompt for fixing bugs.

StarCoder, GPT-4, and Llama 2. We excluded CodeGeeX because its prompt template is rigid, and we could not introduce additional contexts. Due to inferior performance in vanilla prompting, we also excluded the TB-Airoboros and TB-Vicuna. We set the termination criteria so that the mitigation process terminates if the overall increase of successful translation is less than 5%. Figure 6 summarizes our findings. Based on the proposed termination criterion, our mitigation process lasted for two iterations for GPT-4, and one iteration for the rest of the models. The results suggest that the proposed technique can increase the number of successful translations for all the studied LLMs, with $iter_1$—prompting on results from vanilla prompts—increasing the success rate of GPT-4, StarCoder, Codegen, and Llama 2 by 12.33%, 3.55%, 2.65%, 1.97%, respectively. $iter_2$—prompting on results from $iter_1$, can improve GPT-4 by 1.7%.

We also wanted to understand how translation bugs evolve during this iterative prompting process. To that end, we tracked the error outcomes of unsuccessful translations (§4.3.2) from vanilla prompting to $iter_1$ (for all models) and from $iter_1$ to $iter_2$ (for GPT-4). Figure 7 shows our analysis results on GPT-4 (other models' results are available in the artifact [40]).

With $iter_1$, we observe a substantial reduction in *compilation errors* for GPT-4, with 58% completely fixed (no error) and 38% transformed to other translation bugs. However, for other errors, the percentage is lower—45% runtime errors, 48% functional errors, and 43% non-terminating executions—suggesting these bugs are harder to mitigate. With $iter_2$, the outcome of most bugs does not evolve and they remain the same. In both cases, we observe a few cases where the outcome of the translation degrades: i.e., functional error transforms to runtime/compilation error, or runtime error transforms to compilation error. Also, we found several instances where, instead of fixing the bug, the LLM introduced a new one. These findings show that, although the proposed technique improves overall effectiveness, future work should be directed toward combining the power of program analysis and more nuanced LLM approaches, e.g., by

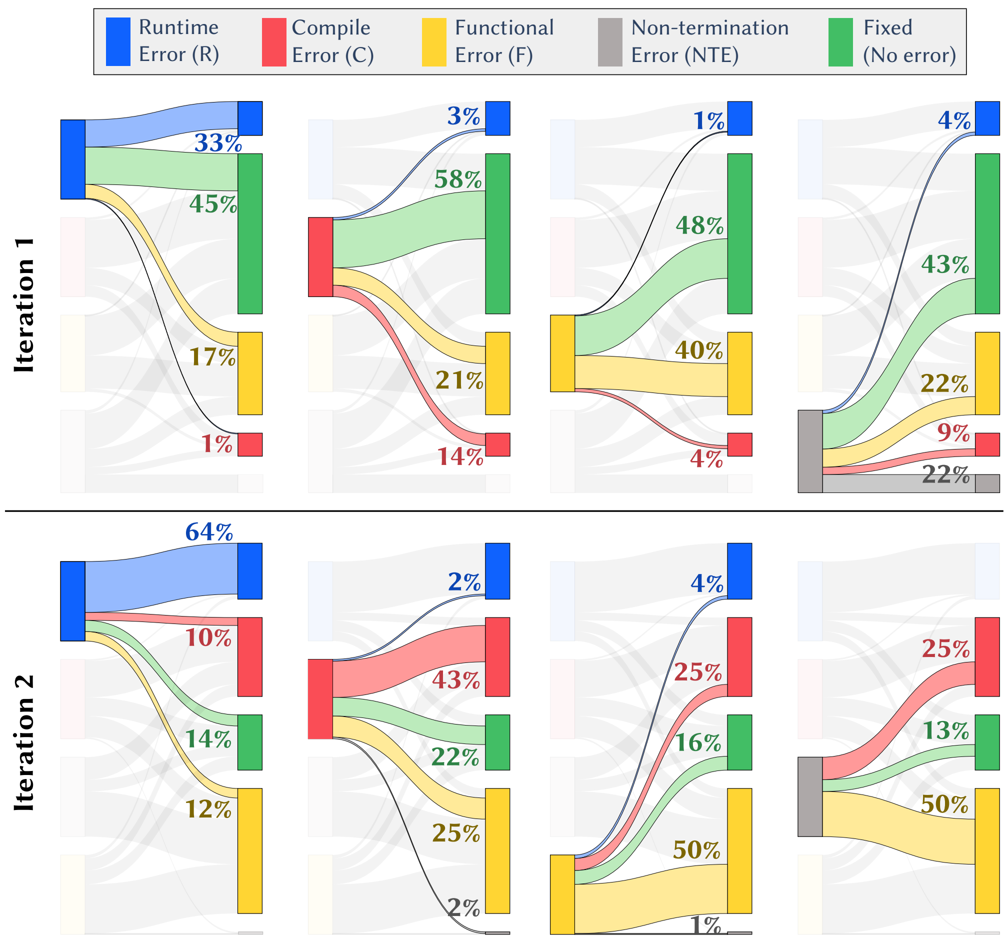



Figure 7: Translation outcomes after prompting GPT-4.

Table 6: Characteristics of LLM-based and non-LLM-based code translation approaches.

| Characteristics | Non-LLM | LLM |
|---|---|---|
| Broader context | ✓ | X |
| Determinism and reasoning | ✓ | X |
| Leveraging target-language idioms | X | ✓ |

including more suitable code examples for in-context learning.

### 4.7 Discussion

***Pros and cons of LLM- and non-LLM-based translation.*** We discuss strengths and weaknesses of each approach with respect to three key characteristics (Table 6). First, non-LLM-based approaches, specifically transpilers, have a more comprehensive context of an application. On the other hand, while translating a module, the variable types, method signatures, folder structure, and dependencies associated with that module are often unavailable to LLMs because of their restriction on the token size. Second, non-LLM-based techniques are generally deterministic, allowing for more predictable reasoning, whereas LLMs are inherently probabilistic, with a greater degree of creativity. The third characteristic pertains to the naturalness of the translated code. In this respect, non-LLM-based approaches mostly do not leverage target language idioms, which can negatively affect code readability and maintainability. In some cases, this even leads to security issues (e.g., C to Rust). Conversely, LLMs tend to generate more natural, human-like code. More research on utilizing both approaches would be fruitful.

***Translating real-world projects.*** Unlike crafted benchmarks, files in real-world projects do not exist as standalone programs, and providing relevant context about inter-file dependencies can help an LLM produce better code. More fundamentally, however, new techniques are required to enable code translation to scale to real-world applications and also generate high-quality translations. For instance, such techniques could use program analysis to provide more context while translating a piece of code.

Another potential direction is to leverage program-decomposition techniques to split the source file into smaller fragments, each translated separately via prompts that encode appropriate context information about the fragment's dependencies; the translated fragments are then composed to produce the fully translated code. Finally, a significant challenge in translating real-world projects is handling library API calls and the differences in the API ecosystem of different PLs. There can be cases where no suitable API exists in the target PL to translate an API call to—techniques that combine code summarization and code synthesis could be investigated to fill such gaps.

***Improving open and closed-source LLMs.*** The findings of this work shows considerable scope for improving open-source and close-source LLMs for code translation. For closed-source models (i.e., GPT-4), the increasing complexity of use cases (i.e., translating real-world projects) calls for an evolution in prompting strategies perhaps with the help of program analysis. One promising research direction could involve creating prompts that build upon each other [163], [164], following a coherent and logical flow of information. For open-source LLMs, enhancing the performance could involve fine-tuning [165], [166], wherein different models [167] (instead of one model fits all) are carefully trained to tackle distinct aspects of the translation process (particularly those related to the bug categories).

## 4.8 Threats to Validity

Like any empirical study, there are threats to the validity of our results. We discuss the most significant ones of these, along with the mitigating factors.

***External Validity.*** Threats to external validity pertain to whether our results can generalize to other experimental settings. Key factors here include the PLs, LLMs, and datasets selected. To mitigate these threats, we selected five PLs, guided by PL popularity [159] while also covering different programming paradigms. In terms of datasets, we used multiple well-known datasets with different characteristics and also two real-world projects. For LLM selection, we included several SOTA general and code LLMs. Finally, for non-LLM-based approaches, we used multiple tools covering different PLs.

***Internal Validity.*** As for threats to internal validity, one potential threat is that, for each translation task, we performed the translation once. Performing a translation task multiple times may change the success rates in translation as LLMs are inherently non-deterministic. However, this does not affect the primary goal of our work, which is to study the characteristics of translation bugs. Regardless of randomness or number of repetitions, the nature of an unsuccessful translation remains unchanged. Moreover, to reduce the effects of LLMs' sensitivity to prompt templates, we followed the best practices described in the respective artifacts/papers/reports (refer §4.2). Another threat to internal validity is that we do not have a formal inter-rater reliability metric for our manual labeling of bugs. To mitigate this threat, after each labeling round, both labelers met to discuss any discrepancies and resolve them. At the end, each bug was assigned a single mutually agreed label. On top of that, one author went through the entire labeling to ensure consistency among the groups. Finally, our results may be affected by bugs in our automation scripts. To address this threat, we thoroughly tested the scripts and spot-checked the results for correctness. Moreover, we make our artifacts publicly available [40] to enable review and replication of our results.

***Construct Validity.*** We measured translation correctness using the test cases provided in our datasets, which is a commonly used approach for assessing the quality of generated code. This approach comes with the risk that inadequate or weak test suites can cause buggy translations that pass the test suites to be considered correct. We note that one of our datasets, CodeNet [130], contains one test case per sample and may be susceptible to this threat. However, the other datasets used contain fairly rigorous test suites.

### 4.9 Related Work

***Code Translation and Synthesis.*** There exist two broad classes of code-translation approaches. The first category includes tools such as transpilers, or source-to-source compilers, that leverage program-analysis techniques for converting code from one PL to another. For instance, C2Rust [18] and CxGo [19] are transpilers that translate C programs to Rust and Go, respectively, whereas Sharpen [20] and Java2CSharp [168] convert Java code to C#. The second category includes learning-based techniques, including lexical statistical machine translation [24], [25] and tree-based neural networks [27] for translating Java to C#. Other work in the category leverages deep learning and unsupervised learning [29], [31] for translating C++, Java, and Python code, phase-based statistical learning [28] for C#-to-Java translation, etc. More recent techniques leverage LLMs (e.g., StarCoder [34], PolyCoder [52], SantaCoder [51], CodeGen [35], BLOOM [54], CodeT5 [53], CodeX [129], GPT-4 [169], Llama 2 [55], etc.) for code generation and code translation (CodeGeeX [36]). However, in the context of LLMs, there is no study that understands the bugs introduced by LLMs during code-translation tasks.

***Bug and repair study.*** Software bugs are well studied [131] including several works on deep learning model-related bugs [132], [133], [134]. Moreover, there is also an extensive list of works on fixing software bugs using LLMs (e.g., [68], [69], [70], [71], and other approaches (e.g., [72], [73], [74]). Compared to both of these classes of studies, here, we study bugs introduced by LLMs while translating code from one PL to another. Also, we compare the effectiveness of LLM with non-LLM-based approaches in terms of code translation task.

### 4.10 Concluding Remarks

Code translation has various applications, from modernizing enterprise applications to migrating legacy software to modern PLs. Given the promising performance of LLMs in code synthesis, we were interested to understand how they perform in code translation task. Our empirical investigation of the general and code LLMs across five PLs and several benchmarks and real-world projects demonstrates that state-of-the-art LLMs are yet to effectively automate code translation, specifically to translate complex real-world projects. Through meticulous manual analysis, we also identified 15 root causes of unsuccessful translations. Furthermore, to investigate how providing more context can help LLMs generate better code, we presented and evaluated an iterative prompt-crafting technique. We also assessed existing non-LLM-based techniques, comparing their strengths and weaknesses. We are currently considering several directions for future work. First, we want to investigate how existing SE/PL techniques can help mitigate these bugs, including giving more context to LLMs. Second, our ultimate goal is to advance real-world application translation.

# Chapter 5 AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation

## 5.1 Introduction

Application modernization offers numerous benefits to developers, including better performance, maintainability, productivity, reliability, and security [1], [2], [3], [170]. Manual migration or modernization of real-world projects can be time-consuming and error-prone. Code translation can help automatically convert programs from one programming language (PL) to another.

Transpilers solely rely on program analysis and perform rule-based translation, failing to translate code between languages that greatly differ in syntax or semantics [23]. This also makes them very PL-specific; they cannot generalize to newer features of the same PL pairs easily, let alone other PLs. Finally, the translations lack readability, requiring much effort to understand and validate them, and naturalness, failing to create idiomatic code in the target PL [22]. State-of-the-art code translation techniques attempt to harvest the emerging abilities of Large Language Models (LLMs) in code synthesis to overcome the limitations of transpilers [22], [84], [85]. However, these techniques are still limited to translating simple programs in crafted benchmarks or selected slices of real-world projects due to the following challenges:

1. *Problem complexity.* The source and target PLs can be fundamentally different in programming paradigms, typing, and memory management. Some PLs have specific properties that may not exist in others, e.g., constructor overloading in Java. Such complexities are beyond the abilities of existing LLMs to handle, causing them to hallucinate when translating types, code constructs, or even method names [22], making translations non-compilable or useless.

2. *Validation.* The translation should preserve the functionality of the source project. Most existing techniques follow a “translation first and validation next” approach,

which can postpone the validation and not benefit from the potential use of validation as feedback to correct the translation [22]. A few techniques use formal methods [84], [85] to verify translations on the go. However, these techniques cannot scale to real-world projects. One possible solution for validation is reusing the tests in the source language. However, due to (1) multiple invocations of different methods in unit tests and (2) inherent long call chains in real-world projects, testing a translated method *in isolation* is impossible.

3. *Limited context window.* Concerning repository-level translation, the entire project and, in many cases, even the entire class cannot fit into the context window of current LLMs [44]. Assuming an unlimited context window, LLMs still suffer from short attention span [38], preventing them from properly capturing the intra- and inter-procedural dependencies in real-world projects.

This paper presents AlphaTrans, a neuro-symbolic[4] approach for automated repository-level code translation and validation. AlphaTrans leverages static analysis to resolve PL-specific features of the source language (§5.4.1), decompose the source project into smaller fragments (§5.4.2), and create a compilable project skeleton in the target language (§5.5). It then starts translating fragments in reverse call order and validates them using existing tests when possible (§5.6). After translating each fragment, AlphaTrans updates the project skeleton and ensures the whole project compiles, gradually translating and validating the source project into the target PL. To improve translation quality, static analysis again comes to the rescue: AlphaTrans collects relevant context for each fragment, including translated callee methods and surrounding contexts, e.g., class declaration, global variables/fields, etc. It also uses relevant in-context examples based on the specific properties of the fragment to be translated. AlphaTrans implements two types of dynamic validation: (1) running the source tests

[4]The keyword symbolic here refers to a general term of symbolic learning in contrast to machine learning and should not be confused with symbolic execution. We refer to combining LLMs and program analysis as a neuro-symbolic approach.

on the translated fragments in isolation using language interoperability (§5.6.1) and (2) decomposing, translating, and executing the source tests on the translated fragments (§5.6.3). Finally, AlphaTrans recomposes the translated fragments to create the program in the target PL (§5.6.2).

Our approach of compositional translation and validation is PL-agnostic; however, implementing the program transformation component is PL-specific. For the first version of AlphaTrans, the implementation supports translating Java code to Python. Our motivations for choosing this PL pair are: (1) Java offers many features that are not supported or common in other PLs by default (e.g., method/constructor overloading, complex types, circular dependencies, local or anonymous inner classes, interfaces, etc.); (2) Python programs are not compiled but interpreted, which makes many translation issues that can be caught at the compile time remain undetected until test execution and challenge the validation; and (3) both PLs are popular (top-5 on the TIOBE index [159]).

Using AlphaTrans to translate ten real-world Java projects to Python corroborates its *effectiveness*: It can translate 17874 field/method/test fragments, with 96.40% syntactic correctness. For the 4654 application method fragments that can be further evaluated through test execution, AlphaTrans achieves 27.03% runtime validation and 25.14% functional equivalence using the source tests. AlphaTrans is *scalable*, completing translations in 34 hours, on average. Human subjects improved partial translation of AlphaTrans and achieved passing test suites within 20.1 hours, on average, showing *practicality* of AlphaTrans. These results were achieved using a moderate-size open-access LLM (DeepSeek-Coder-33b-Instruct [37]). A stronger model, i.e., GPT-4o, improves the performance of AlphaTrans to 99.2% syntactic correctness for all fragments and 27.95% functional equivalence for method fragments, with an overhead of $14.39 per project, on average. The affordable cost is due to the novel features of the pipeline, namely, decomposition into fragments, prompt crafting, in-isolation validation of translations, and efficient feedback loop.

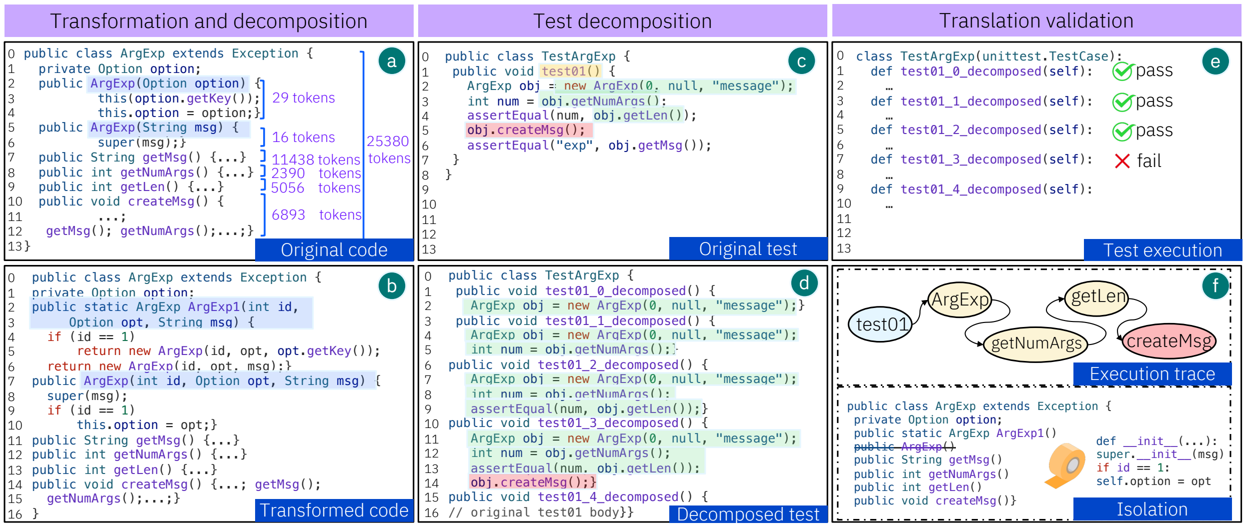



Figure 8: Illustration of key challenges in repository-level code translation and AlphaTrans addressing them.

To the best of our knowledge, AlphaTrans is the *first technique to translate an entire repository*, including tests, and generates validated translations (considering existing tests). The only prior repository-level translation attempt using GPT-4 [22] (translating Apache Commons CLI from Java to Python) resulted in non-compilable code, let alone the translation being validated. AlphaTrans is also *the first technique leveraging language interoperability for in-isolation validation of translated fragments.*

The effort of human subjects to fix translation bugs by AlphaTrans and achieve passing tests creates pragmatic bug data sets for testing, bug localization, and program repair research. Our code and artifacts are publicly available for reproducing the results or translating new projects [41].

## 5.2 Challenges in Repository-Level Code Translation

To illustrate the most notable challenges in repository-level code translation and validation, we use the hypothetical example in Figure 8, inspired by the complexities in real-world Java projects.

**Challenge 1: Class Size.** The class consists of 25,380 tokens ( a ). Instructions for translating the code, in-context examples, and the model's response can also significantly increase the number of input tokens. While some commercial LLMs support tens of thousands of tokens, many open-access LLMs do not. For example, DeepSeek-Coder-33b-Instruct [37] used in this paper has a context window of $16,384$ tokens, of which only $4,096$ tokens can be used for generation. To address this challenge, AlphaTrans decomposes Java application classes into smaller *field* and *method fragments* and translates each separately in reverse call order.

**Challenge 2: PL-specific Properties.** Java programs frequently use method/constructor overloading, which is not supported directly in Python ( a ). This example shows instances of constructor overloading (lines 2 and 5). In Python, declaring two constructors is allowed; however, at runtime, the last declaration overrides all previous constructors, resulting in unexpected behavior. To address this issue, AlphaTrans employs program analysis to refactor the original code while preserving the functionality (through test execution). The transformation includes changing the constructor's name, updating the references, and changing the constructor's implementation. The transformed code ( b ) makes the source program amenable to translation to Python.

**Challenge 3: Validation.** To illustrate the challenges with validation, consider `test01` ( c ) that invokes *four* methods in its body (`ArgExp`, `getNumArgs`, `getLen`, and `createMsg`) to test the functionality of method `getMsg` in the assert statement. Suppose we can successfully translate all methods except `createMsg`. If we choose test translation (a natural way of validating code translation), the execution of the translated test results in a runtime error when invoking `createMsg`. As a result, a translation issue in one method casts a shadow in validating the translation of the other methods. We refer to this issue as the *test translation coupling effect*. To overcome this challenge, AlphaTrans executes source language tests as-is (i.e., without translation) by leveraging a language-interoperability framework called GraalVM [91] ( f ). In this setting, a test in the source language is executed each time one of its invoked application methods (method fragments) is translated. This approach validates *functional equivalence* of each method *in isolation* as the other invoked application methods during test execution remain in the source language.

**Challenge 4: Test Translation.** GraalVM has certain limitations (§5.6.1), which prevents AlphaTrans from validating all the code fragments in isolation. Furthermore, we need to translate tests regardless of whether they are used for validation to maintain the translated projects in the target language. Test errors due to test translation coupling effect under-approximate the quality of translation: failing to validate the translation of four methods because of one incorrect translation. To overcome this challenge, AlphaTrans decomposes the original test suite into *test fragments* ( d ). Executing the translated decomposed test suite results in three test passes ( e ), validating the *runtime behavior* of three methods that the original test suite could not promptly provide.

An alternative approach is parsing the stack trace and code coverage results for each runtime error during translation. However, test decomposition is a cleaner way to see the results per test execution promptly. It is also done once before translation. In translation to interpreted languages such as Python, specifically, the execution of test fragments can validate the runtime behavior of methods before waiting for functional validation. For fragments that GraalVM cannot validate, if AlphaTrans can successfully translate all the methods invoked during test execution and the test passes, such a test will also be used for validating *functional correctness*.

## 5.3 Overview of Approach

AlphaTrans consists of three main phases, shown in Figure 9: program transformation and decomposition (§5.4), type translation and skeleton construction (§5.5), and compositional translation and validation (§5.6). The first two phases aim to decompose and simplify the repository-level code translation problem for LLMs, helping the third phase yield high-quality validated translations.

The program transformation and decomposition phase first refactors the PL-specific properties of the source program into programming paradigms common among many PLs (§5.4.1). Next, it decomposes the source project into smaller units, i.e., *fragments*, and stores fragment dependencies in a data structure called *schema* (§5.4.2).

The type translation and skeleton construction phase takes the schema as input and produces *target project skeleton*, i.e., a compilable project in the target language with method signatures but no method implementation (§5.5.2). The first translation step happens here, where the source PL types are translated to the target PL to ensure that class skeletons are compilable (§5.5.1). The outcome of type translation is a type mapping from the source to the target PL, which AlphaTrans can reuse in translating other projects.

The compositional translation and validation phase takes the schema and project skeleton as inputs and translates fragments, in reverse call order, by prompting an LLM. After translating a fragment, it updates the class skeleton with a new translation and checks whether the skeleton compiles. For a method fragment, AlphaTrans looks for corresponding tests and, if any exist, uses them to validate the fragment. The first level of validation is performed through GraalVM's language interoperability to isolate the validation of the method using tests in the source language. Next, AlphaTrans translates and executes the tests. In case of compilation errors or test failures, AlphaTrans reprompts the LLM with feedback (from the compilation/runtime errors) to improve the translation. If no improvement is achieved within a certain budget, AlphaTrans continues to the next fragment until all are translated. For methods whose translations are not compilable or result in test errors/failures, AlphaTrans generates reports consisting of existing translations and relevant artifacts, such as stack traces, test errors/failures, and test coverage information.

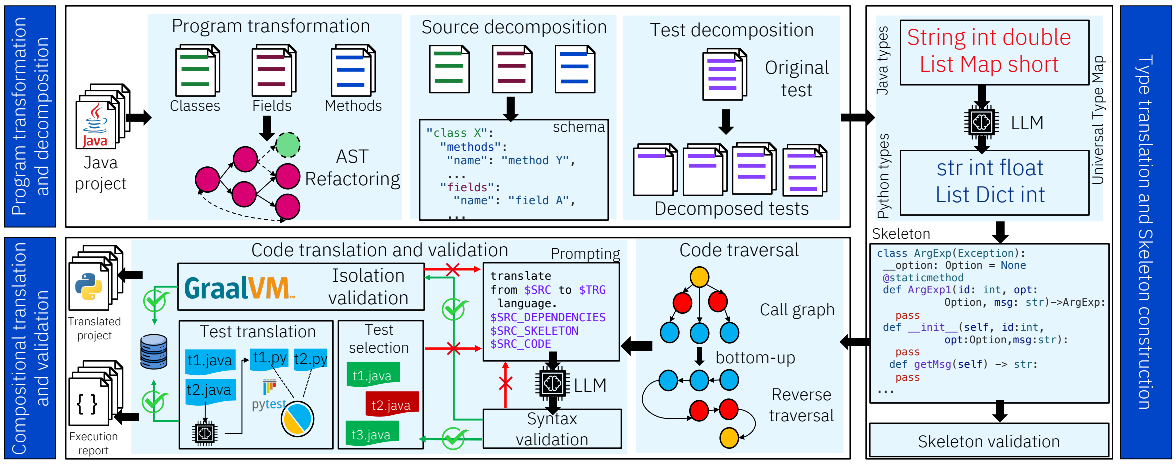



Figure 9: Overview of AlphaTrans.

### 5.4 Program Transformation and Decomposition

#### 5.4.1 Program Transformation

This component performs semantics-preserving refactoring of method and constructor overloading in Java code to make it amenable to translation to Python. Other Java-specific features, namely, circular dependencies, inner classes, interfaces, and abstract classes, are handled while constructing the project skeleton in Python (§5.5.2). The reason for resolving method and constructor overloading in the source language is that we have to change the implementation, i.e., call sites to methods and constructors. Therefore, such changes should be validated using source tests before translation.

For overloaded methods, AlphaTrans makes each method name unique by adding an integer suffix (starting at 0) to the name, and updates all call sites based on the new method names. Resolving overloaded constructors is not as straightforward, as they should have the same name as the enclosed declaring class. Furthermore, the invocation of constructors inside each other and the Java inheritance mechanism makes constructor overloading complex. Our algorithm (Algorithm 1) for resolving the constructor overloading handles three prominent patterns shown in Figure 10.[5]

**Algorithm 1** Constructor Overloading

```
Require: OCs                                        ▷ Overloaded Constructors
Ensure: NOCs                          ▷ Code without Overloaded Constructors

if not hasThisCall(OCs) then
    ids ← createConstructorIDS(OCs)
    NOCs ← merge(OCs, ids)
else
    if hasOnlyThisCall(OCs) then
        refactor ← createRefactorMethod(OCs)
        NOCs ← merge(OCs, refactor)
    else
        refactor ← createRefactorMethod(OCs)
        ids ← createConstructorIDS(OCs)
        NOCs ← merge(OCs, refactor, ids)
    end if
end if
refactorCallSites()
return NOCs
```

The first pattern (Figure 10-a) shows multiple independent constructors. AlphaTrans merges these constructors into one and uses an `id` parameter to differentiate between them. All call sites of the constructors are updated accordingly to use the appropriate `id` value. The second pattern (Figure 10-b) involves a constructor call chain using `this()`. AlphaTrans transforms the first constructor into a factory method and invokes the second constructor inside it. Factory methods are static, and AlphaTrans updates the call sites to invoke them directly on the class, e.g., `ArgExp.ArgExp1(id,opt,msg)`. The last pattern (Figure 10-c) is similar to the second one, except that both constructors implement some code. AlphaTrans refactors the first constructor into a factory method and adds an extra `id` parameter to differentiate between behaviors implemented by different constructors. The constructor call sites are updated accordingly, as in the previous cases. Real-world projects often combine these patterns, which AlphaTrans handles using Algorithm 1.

[5] Following the best practices for constructor overloading from *Stack Overflow* and analyzing the use of constructor overloading in open-source projects, we categorized the use cases into the three patterns.

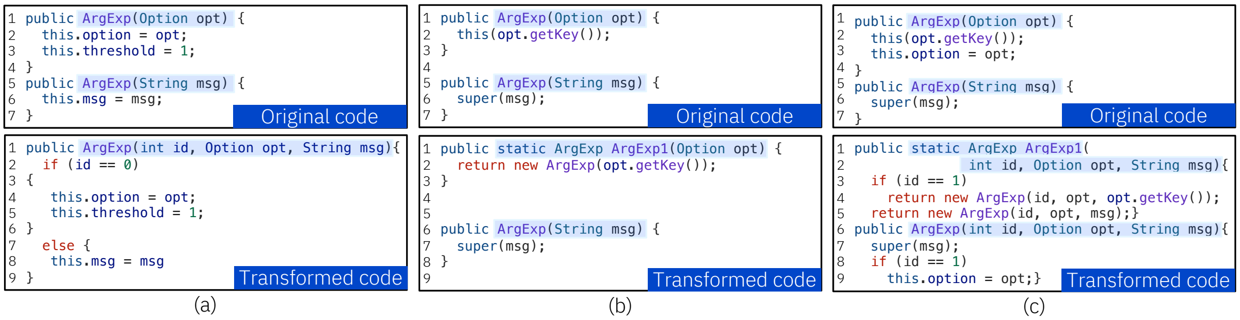



Figure 10: Constructor overloading patterns and their corresponding transformations.

### 5.4.2 Program Decomposition

Translating the entire repository of real-world projects is a very complex problem. As a result, AlphaTrans breaks down projects into fragments, performs the translation and validation at the fragment level, and re-composes the translation as a repository in the target language.

**Source Decomposition.** Real-world projects can include hundreds of files with thousands of lines of code, which exceed the context window of state-of-the-art LLMs. AlphaTrans employs static analysis to decompose code into smaller fragments, i.e., *field fragments* and *method fragments*. A field fragment includes modifiers, type, name, and field value. A field fragment can belong to an application or test class. A method fragment includes the method signature and can be an application or test method (e.g., helper methods or unit tests). During decomposition, AlphaTrans extracts meta-information related to the fragments, such as their location (e.g., start and end line), code (e.g., implementation between start and end line), dependencies (e.g., callers and callees), types (of inputs and output), and other necessary information such as file paths, class inheritance, imports, and method annotations. AlphaTrans stores fragments and their corresponding collected meta-data in a data structure called *schema*, which is used in the other phases. AlphaTrans also extracts the call graph to guide the translation, i.e., to translate fragments in reverse call order.

**Test Decomposition.** The burden of checking *functional equivalence* in AlphaTrans is on GraalVM. Yet, we still need to translate and execute tests to validate the fragments that cannot be validated by GraalVM (§5.6.1). Unit tests in real-world projects can invoke multiple methods and include multiple assert statements. Furthermore, long call chains are inevitable in real-world projects due to the high degree of intra- and iter-procedural dependencies. As we show later (§5.7.4), the average number of direct method invocations and method executions in tests for our studies subjects are 3 and 27, respectively. This can result in *test translation coupling effect*, discussed in §5.2.

To enable test translation for runtime validation or checking functional equivalence, AlphaTrans decomposes each unit test into a series of *test fragments*, as shown in Figure 8-d. It uses each statement with a call to an application method as a cutting point. For statements enclosed by branches, loops, of exception-handling blocks, AlphaTrans includes the entire block. This process generates an ordering of executable test fragments for each unit test. Each test fragment includes all the statements of the lower-order fragments, along with additional statements that invoke one additional method not invoked by previous fragments. AlphaTrans executes test fragments in increasing order until a test fails and skips running following fragments, as they will also fail.

## 5.5 Type Translation and Skeleton Construction

### 5.5.1 Type Translation

Automatically resolving types is a challenging problem [171], [172], and a large body of work has attempted to address this, mostly using symbolic rule-based approaches [29], [173], [174], [175], [176], [177]. AlphaTrans employs a Retrieval-Augment Generation (RAG) [178] technique for finding equivalent types in the target language. To that end, it first extracts all the types in the source language of a given project. Custom application types are resolved during the translation as AlphaTrans translates fragments and

```
class ArgExp(Exception):
  __option: Option = None
  @staticmethod
  def ArgExp1(id: int, opt: Option, \
              msg: str) -> ArgExp:
    pass
  def __init__(self, id: int, opt: \
               Option, msg: str):
    pass
  def getMsg(self) -> str:
    pass
  def getNumArgs(self) -> int:
    pass
  def getLen(self) -> int:
    pass
  def createMsg(self) -> None:
    pass
```

Figure 11: Target PL skeleton for the example in Figure 8-b.

classes in the target language. For the remaining types, it crawls the API documentation of the source language and retrieves the relevant description of each type.

To form the prompt,[6] AlphaTrans uses the retrieved description and instructs the model with an in-context example to return the most relevant type in the target language, given the use of types in the source language and the retrieved description. To account for potential hallucination in LLM's response, i.e., returning a type that does not exist in Python, AlphaTrans employs a simple Python script, uses the translated type as an annotation, executes the script, and keeps the ones that have no syntactic or runtime issue. The types in the source language and their corresponding in the target language form a data structure called *universal type mapping*. In practice, AlphaTrans reuses or augments the mapping when translating new projects.

### 5.5.2 Skeleton Construction

AlphaTrans builds the project's structure in the target language before translation. This step is necessary for compositional translation and validation, as AlphaTrans can

[6]Due to space limit, we omit the prompt; please refer to our artifacts [41] to see the prompts used for type resolution.

insert the translated fragments into the project, compile it, or even execute the existing translated test suites, gradually completing the translation. At this step, AlphaTrans also resolves Java-specific features in Python before starting the translation. Specifically, it resolves circular imports and dependencies, inner classes, interfaces, and abstract classes. Figure 11 shows the class skeleton corresponding to the illustrative example of Figure 8-b.

At the first step, AlphaTrans creates Python classes corresponding to each application class in Java. The fields in the classes are set to `None`, and AlphaTrans uses the information in schema to ensure the naming corresponds to the type of their access modifier in the source language. In the example of Figure 11, the translation of field `private Option option;` in Java is `__option: Option = None`. The classes also include method signatures, with their body set to `pass`. AlphaTrans uses the universal type mapping (§5.5.1) to create relevant types in the method signature. Once the initial skeleton is created, AlphaTrans leverages the extracted call graph during program decomposition to detect circular dependencies. If such dependencies exist, AlphaTrans resolves them with local imports. For inner classes, AlphaTrans unfolds them in Python and uses *dot notation* to access specific methods and fields (e.g., `Class.methodName`). Finally, AlphaTrans implements best practices in Python and subclasses all abstract classes and interfaces from `abc.ABC` class. Here, `ABC` is a class from the `abc` module in the Python standard library, which is used for defining abstract base classes.

### 5.6 Compositional Translation and Validation

AlphaTrans translates method fragments in reverse call order. It takes the project's call graph, removes the back edges to make it acyclic, computes topological order (i.e., linear ordering of its vertices), and translates method fragments (corresponding to vertices) in reverse topological order. Field fragments are independent; therefore, AlphaTrans translates them before method fragments. The algorithm takes a fragment

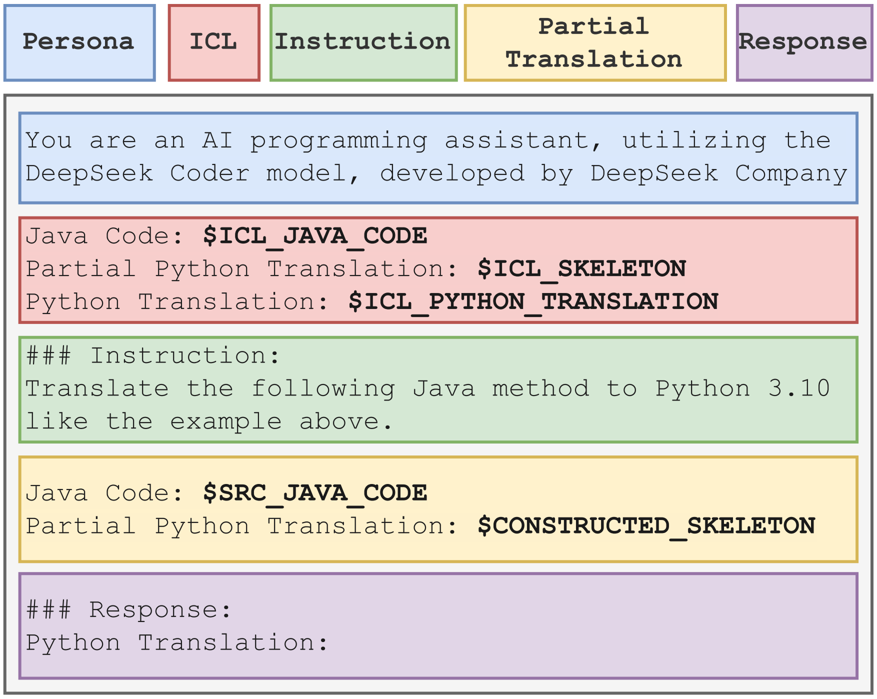



Figure 12: Prompt structure in AlphaTrans.

$F$, LLM $M$, Skeleton $S$, and a series of parameters as inputs, translates the fragment, recomposes the skeleton with the successfully translated fragment, and returns translation outcomes: (1) syntax check (denoted by the "non-parseable" and "parseable" labels), (2) functional equivalence check (denoted by the "graal-fail", "graal-success", and "graal-error" labels), and (3) translated test execution check (denoted by "not-exercised", "test-fail", and "test-success" labels).

AlphaTrans employs iterative and feedback-based prompting. That is, if one of the mentioned checks fails, e.g., the translated fragment is not syntactically correct, it prompts the model for another translation. To control the number of iterations, AlphaTrans uses a reprompting budget (*repromptBudget*). The algorithm takes the minimum ($min_{budget}$) and maximum ($max_{budget}$) values for the budget and dynamically sets reprompting budget to a number between the lower and upper bounds based on coverage information, e.g., the budget is close to $max_{budget}$ for a fragment if it is exercised multiple times (high hit rate based on coverage information) by different unit tests. The rationale is to give more importance to fragments covered by more tests to eventually increase translation validation success.

Algorithm 2 shows the main translation and validation loop (lines 3–35), which runs until the reprompting budget is exhausted. Inside the loop, AlphaTrans first crafts a unique prompt based on the template shown in Figure 12 and then instructs the LLM to translate the fragment (lines 4–5). It then validates the generated translation in multiple steps. The first step checks for syntactic correctness and assigns proper labels to $TVO$ (lines 6–10). Then, AlphaTrans leverages GraalVM for isolation-based validation of fragment $F$ (lines 13–22), if there exists a test in the source language covering the fragment during its execution.

Finally, it translates and executes decomposed fragment tests: if there are no eligible tests (a test becomes eligible if all its dependencies are translated) for the fragment, AlphaTrans simply assigns the "*not-exercised*" label to the fragment and moves on to the next one (lines 23–26). Otherwise, it translates the tests, executes them to validate the fragment, and assigns test outcome labels in $TVO$ (lines 27–35). In case of a test failure, AlphaTrans extracts all involved fragments and reprompts them with feedback extracted from test execution.

Due to inherent intra- and inter-procedural dependencies in real-world projects, the number of fragments involved in reprompting could be high, logarithmically increasing the translation time. AlphaTrans filters out those with GraalVM label "*graal-success*", ranks the remaining based on suspiciousness score, and reprompts *topK* suspicious fragments. The suspiciousness score for fragments is calculated such that a fragment with more failing tests will get a higher score and, therefore, ranked higher among other fragments.

Our prompt template consists of *five* distinct parts (Figure 12). The first part is the *persona* message used by DeepSeek-Coder-33b-Instruct during instruction fine-tuning and is required for producing the best outputs. The next part introduces the *In-Context*

**Algorithm 2** Main Translation Loop

**Require:** Skeleton $S$, Fragment $F$, Model $M$, $min_{budget}$, $max_{budget}$, $topK$ suspicious methods
**Ensure:** Translation and Validation Outcome $TVO$

```
feedback ← ∅; TVO ← {}
repromptBudget ← getAdaptiveBudget(F, min_budget, max_budget)

while repromptBudget > 0 do
    prompt ← generatePrompt(F, feedback)
    translation ← translateFragment(prompt, M)

    if not syntacticCheck(translation) then
        TVO["syntax_outcome"] ← "non-parseable"
        feedback ← getFeedback(translation)
        repromptBudget ← repromptBudget − 1
        continue
    end if
    TVO["syntax_outcome"] ← "parseable"

    if not graalCheck(translation) then
        TVO["graal_outcome"] ← "graal-fail"
        feedback ← getFeedback(translation)
        repromptBudget ← repromptBudget − 1
        continue
    else if graalLimitation(translation) then
        TVO["graal_outcome"] ← "graal-error"
    else
        TVO["graal_outcome"] ← "graal-success"
    end if

    if not hasEligibleTests(F) then
        TVO["test_outcome"] ← "not-exercised"
        break
    end if
    testTranslation ← getTestTranslation(F)
    S ← S ∪ translation

    if not testCheck(S, testTranslation) then
        TVO["test_outcome"] ← "test-fail"
        repromptSuspiciousMethods(testTranslation, topK)
        repromptBudget ← repromptBudget − 1
        continue
    end if
    TVO["test_outcome"] ← "test-success"
    break
end while

return TVO
```

*Learning (ICL)* example, which reflects the complexities of code translation and instructs LLM on how to deal with them. The green part shows the natural language instruction given to the model. After describing the objective, the prompt embeds the source Java code along with *partial translation* as a skeleton, which includes all dependencies and translations of the fragments invoked by the current one. The prompt concludes with `### Response:` keyword to guide the model for code generation.

### 5.6.1 Language Interoperability

GraalVM [91], [179] is a Java Development Kit by Oracle. It offers the *Polyglot* API [180], which allows the integration of programs written in different guest languages within a Java-based host application. In the context of this work, GraalVM allows execution of Python code from Java and vice versa [92]. AlphaTrans leverages the Polyglot API to perform *in-isolation* validation of the fragments by replacing the Java implementation of a method with its translated Python version while keeping the rest of the project in Java. It then executes the Java tests covering the fragment to validate the functional equivalence of the translation.

The Polyglot API allows access to Python data objects from Java and vice-versa, as these objects reside in a shared memory space. However, objects must be cast to appropriate types for passing parameters to, and processing returned values from, polyglot calls. The Polyglot API can perform this casting implicitly for only a few simple data types. AlphaTrans builds on top of the Polyglot API to provide a framework to create a Python program state that is isomorphic to the Java program state. The Python translation is restricted to this isomorphic state, and the states are synchronized after method calls to preserve the isomorphism. AlphaTrans allows for the casting of user-defined types as well as several built-in and library types. Using both static and dynamic type information of Java objects, AlphaTrans can disambiguate target types when casting Python objects to Java types. It further preserves object identities and aliasing during such casting and propagates exceptions across language boundaries.

To validate the translation of a method $m$ in isolation, AlphaTrans creates an instrumented version of the Java source code. We refer to this instrumented Java project as the *primal project*, $P_m$. During instrumentation, AlphaTrans replaces the original Java implementation, $m_J$ of $m$ with a polyglot call to its Python implementation, $m_P$. $m_P$ resides inside a Python project, which we refer to as the *dual project*, $D_m$. The structure of $D_m$ is similar to that of the original Java project. All other methods in $D_m$ wrap a call to the corresponding methods in $P_m$. Doing so provides an interface for $m_P$ to execute with access to all other methods and fields, although these are defined only in $P_m$. Using the call graph for the Java project, AlphaTrans determines all test methods that invoke $m$ and executes them in $P_m$ to validate the translation $m_P$. This *in-isolation* validation approach is limited in the sense that it can handle only a limited number of built-in and library types. In certain cases, such as reference cycles involving maps or objects with impure methods for hashing, the isomorphism between Java and Python states may not be maintained. Furthermore, it may sometimes not be possible to disambiguate target types when casting Python objects to Java types, for example, if the target object has type `List<Object>`.

### 5.6.2 Target Program Recomposition

AlphaTrans recomposes the project skeleton with syntactically correct translated fragments at each iteration of Algorithm 1, gradually constructing the project in the target PL. The recomposed target program in Python is executed against eligible translated tests, i.e., those that cover only the translated fragments.

### 5.6.3 Test Translation

Similar to translating application fragments, AlphaTrans also translates test fragments. Using the dependency information captured during static analysis, it crafts prompts for unit tests along with their dependencies for the model to translate. The ICL examples used for prompting test fragments differ from prompts used for translating application method fragments. The focus of ICL examples here is to prevent the LLM from hallucinating the used assert statements in the source PL to the target PL. To construct ICL examples for test fragment translation, we created a pool of in-context examples, where each example shows the Python assert statements equivalent to Java assert statements in the context of a test. When prompting a test fragment, AlphaTrans detects the assert statement in the fragment and retrieves the corresponding examples from the pool. For translated tests, only syntactic validation is performed as there is no other means of validating their translations.

### 5.7 Empirical Evaluation

To evaluate different aspects of AlphaTrans, we investigate the following research questions:

**RQ1:** *Effectiveness of AlphaTrans.* To what extent AlphaTrans can automatically resolve types from source to target PL? Can AlphaTrans effectively translate real-world projects?

**RQ2:** *Translation Bugs and Fixes.* How much effort do developers spend completing the partial translations by AlphaTrans? What is the nature of translation bugs?

**RQ3:** *Impact of Test Decomposition.* What is the impact of test decomposition on validation results?

**RQ4:** *Impact of Test Coverage.* To what extent does a test suite with higher coverage impact the validation results?

**RQ5:** *Ablation Study.* To what extent do program transformation, choice of LLM, and program decomposition impact the performance of AlphaTrans?

### 5.7.1 Experiment Setup

We followed three steps for selecting subjects: **1- Mining:** We mined GitHub and retrieved a list of repositories that use Java as the primary language, are self-contained (include build files, etc.), and have more than 30 stars with at least one commit pushed within the last 12 months. **2- Filtering:** We filtered out projects based on the number of call edges in their call graphs: removed those with less than $2,000$ call edges to ensure the subject projects are big enough to challenge AlphaTrans. We also removed those with more than $30,000$ call edges to reduce the computation and manual effort for further steps. Per GraalVM requirements, we only selected projects we could successfully build (compile and achieve green tests) using Java at 21. **3- Reduction:** AlphaTrans currently supports the following Java APIs: core Java API (`java.util`, `java.text`, `java.lang`, `java.io`, `java.nio`, `java.net`, `java.time`, and `java.math`) and third-party libraries (`org.opentest4j`, `org.slf4j.Logger`, and `org.junit`). We automatically removed all other third-party library dependencies and their usage in the source code in the selected projects. We chose a project if at least 50% of its total methods were preserved after such process. Table 7 shows the list of ten projects used in the evaluation of AlphaTrans and details about their size (classes, methods, tests, and fragments).

AlphaTrans uses CodeQL [181] and tree-sitter [113] for static analysis. For running tests, validating translation, and computing coverage, AlphaTrans uses GraalVM $21.0.3 + 7.1$ [91], JUnit 4 and 5 [182], Pytest 8.2.1 [183], JaCoCo [184], and Python's `coverage` [185]. AlphaTrans works with API- and open-access LLMs. We considered the following criteria for selecting the LLM: (1) for better reproducibility, we prioritized open-access models; (2) due to computing constraints, we wanted an LLM with moderate size ($> 10B$ but $< 70B$ parameters); (3) the model should perform reasonably well in code-related tasks; and (4) the model should have fast inference time due to the huge number of prompts. Per the mentioned criteria, we selected DeepSeek-Coder-33b-Instruc [37] for the main experiments (RQ1–RQ4). We also used

Table 7: Effectiveness of AlphaTrans in program transformation, automated type translation, and skeleton validation. **ATR:** Automated Types Resolution, **SV:** Skeleton Validation. The number of classes and methods include both application and test classes/methods.

| Subjects | # Classes | # Methods | # JUnit Tests | Method Coverage (%) | ATR (%) | SV (%) | # Fragments | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | Fields | | Application Methods | Test Methods |
| | | | | | | | Application | Test | | |
| cli [186] | 58 | 664 | 437 | 94.14 | 96.60 | 100 | 104 | 57 | 273 | 2180 |
| codec [187] | 156 | 1780 | 992 | 91.03 | 96.01 | 100 | 425 | 140 | 680 | 2849 |
| csv [188] | 41 | 694 | 309 | 90.64 | 92.34 | 100 | 146 | 35 | 235 | 1272 |
| exec [189] | 56 | 407 | 70 | 54.84 | 78.90 | 100 | 104 | 27 | 248 | 327 |
| fast-pfor [190] | 82 | 971 | 82 | 54.55 | 87.40 | 100 | 127 | 14 | 748 | 302 |
| fileupload [191] | 49 | 381 | 39 | 13.02 | 98.31 | 100 | 121 | 39 | 192 | 194 |
| graph [192] | 118 | 879 | 146 | 58.78 | 97.13 | 100 | 216 | 29 | 541 | 975 |
| jansi [193] | 48 | 474 | 107 | 23.47 | 84.83 | 100 | 378 | 0 | 409 | 123 |
| pool [194] | 98 | 1097 | 73 | 22.29 | 91.60 | 100 | 203 | 91 | 682 | 649 |
| validator [195] | 130 | 1228 | 464 | 63.31 | 95.51 | 100 | 421 | 209 | 646 | 1463 |
| **Total** | 836 | 8575 | 2719 | 56.57 | 91.99 | 100 | 2245 | 641 | 4654 | 10334 |

GPT-4o, one of the best-performing commercial models, for RQ5 (§5.7.6) to demonstrate the impact of stronger models on improving the performance of AlphaTrans. We prompted models with the temperature set to 0 for reproducibility and used their default settings for other parameters. For the base prompting (Algorithm 2), we set the minimum and maximum values of the reprompting budget to 3 and 5. For the feedback prompting, we set the reprompting budget to 1, i.e., AlphaTrans attempts to fix issues with feedback only once.

### 5.7.2 RQ1: Effectiveness of AlphaTrans

In this RQ, we evaluate AlphaTrans in (1) type translation and skeleton construction and (2) compositional translation and validation.

**Effectiveness in Type Resolution and Skeleton Validation.** AlphaTrans extracted $1,797$ distinct types from the source projects and attempted to translate them to equivalent Python types. Of these, 915 are application types (i.e., classes defined within the Java projects) and were directly resolved during skeleton construction. AlphaTrans prompts DeepSeek-Coder-33b-Instruct to resolve the remaining 882 and successfully translated 738 ($(915 + 738)/1,797 = 91.99\%$) of them: the generated types passed the syntactic and runtime check. The column *ATR* in Table 7 shows the results of automated type resolution. Because type resolution is essential to project skeleton construction, we manually checked the type mappings generated by DeepSeek-Coder-33b-Instruct and also translated the 144 unresolved types.

Through manual investigation of the automatically resolved types, we observed that DeepSeek-Coder-33b-Instruct's type resolution for 182 cases, while *correct*, can be *improved*. For example, AlphaTrans translated `java.io.File`, a class concerning file manipulation functionality to `str`. The resolved type can represent file paths in Python but lacks features for file manipulation. We suspect this translation is impacted by the Java use case provided in the prompt. While this translation is correct for the use case, we replaced it with `pathlib.Path` to have a more generic type mapping. We also augmented the type mapping with additional types in the target language for 38 types. For example, AlphaTrans translated `java.nio.Buffer` to `bytearray`, which is correct as they both provide a mutable sequence of bytes with efficient in-place modifications. However, `array.array` and `memoryview` also provide similar functionality with efficient and low-level data manipulation capabilities. Consequently, we augmented type mapping to `typing.Union[bytearray, array.array, memoryview]`. Given that type mapping can be reused, this one-time manual effort increases the chance of AlphaTrans's success on unseen projects.

Using the universal type mapping, AlphaTrans successfully creates and validates project skeletons in the target PL, achieving 100% syntax and runtime validation (column $SV$ in Table 7). The skeleton validation step ensures all module imports, class structures, method signatures, and type annotations are done properly, making the subsequent steps easier. Applying AlphaTrans to unseen projects, if a class skeleton cannot be validated, AlphaTrans removes it from the target project, updates the skeleton based on the class dependencies, and proceeds to the next phase.

Table 8: Effectiveness of AlphaTrans in repository-level code translation. Abbreviations in the table stand for **AMF**: #Application Method Fragments, **SNEF**: Source Non-Exercised Fragments, **GS**: Graal Success, **GF**: Graal Fail, **GE**: Graal Error, **TNEF**: Target Non-Exercised Fragments, **ATP**: Fragments All Test Pass, **OTF**: Fragments One Test Fail, **MTF**: Fragments Many Test Fail, **ATF**: Fragments All Test Fail, **TPR**: Test Pass Rate, **O**: Overall, **RE**: Runtime Error, **AF**: Assertion Failure, and **M1**: Number of *AMFs* that GraalVM could not execute (*GE*) but translated test fragments exercised.

| Subjects | AMF | Syntax Check (%) | SNEF (%) | GraalVM | | | Test Translation | | | | | | | | | | | | M1 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | GS (%) | GF (%) | GE (%) | TNEF (%) | ATP (%) | OTF (%) | | | MTF (%) | | | ATF (%) | | | TPR (%) | | |
| | | | | | | | | | O | RE | AF | O | RE | AF | O | RE | AF | | All | Some |
| cli | 273 | 100 | 5.86 | 70.70 | 11.72 | 11.72 | 35.90 | 8.42 | 10.62 | 51.72 | 48.28 | 32.23 | 91.07 | 8.93 | 6.96 | 100 | 0 | 10.08 | 0 | 16 |
| codec | 680 | 98.53 | 8.97 | 38.38 | 32.50 | 20.15 | 65.59 | 4.12 | 4.41 | 60.00 | 40.00 | 12.79 | 55.11 | 44.89 | 4.12 | 75.21 | 24.79 | 9.43 | 11 | 27 |
| csv | 235 | 98.72 | 9.36 | 38.72 | 26.81 | 25.11 | 74.47 | 0 | 13.62 | 96.88 | 3.13 | 0 | 0 | 0 | 2.55 | 100 | 0 | 0 | 0 | 3 |
| exec | 248 | 100 | 45.16 | 33.47 | 2.02 | 19.35 | 34.27 | 4.44 | 2.02 | 40.00 | 60.00 | 7.26 | 93.36 | 6.64 | 6.85 | 100 | 0 | 19.29 | 6 | 9 |
| fast-pfor | 748 | 95.32 | 45.45 | 12.03 | 24.20 | 18.32 | 41.71 | 4.28 | 1.74 | 84.62 | 15.38 | 3.74 | 79.23 | 20.77 | 3.07 | 85.88 | 14.12 | 20.08 | 6 | 25 |
| fileupload | 192 | 100 | 86.98 | 8.85 | 1.04 | 3.13 | 1.56 | 3.65 | 6.77 | 30.77 | 69.23 | 1.04 | 91.67 | 8.33 | 0 | 0 | 0 | 63.44 | 2 | 3 |
| graph | 541 | 99.63 | 41.22 | 24.77 | 22.92 | 11.09 | 57.12 | 0 | 0.92 | 100 | 0 | 0.18 | 100 | 0 | 0.55 | 100 | 0 | 11.04 | 0 | 1 |
| jansi | 409 | 99.76 | 76.53 | 8.07 | 11.49 | 3.91 | 22.25 | 0.24 | 0.98 | 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.07 | 0 | 1 |
| pool | 682 | 100 | 77.71 | 6.01 | 1.32 | 14.96 | 19.35 | 1.61 | 1.03 | 100 | 0 | 0.29 | 100 | 0 | 0 | 0 | 0 | 6.62 | 4 | 2 |
| validator | 646 | 99.23 | 36.69 | 30.50 | 11.15 | 21.67 | 46.44 | 3.25 | 5.42 | 71.43 | 28.57 | 4.18 | 84.50 | 15.50 | 4.02 | 99.12 | 0.88 | 11.70 | 1 | 31 |
| **Total** | 4654 | 98.80 | 43.43 | 24.50 | 16.23 | 15.84 | 41.92 | 2.88 | 3.72 | 70.52 | 29.48 | 5.44 | 82.77 | 17.23 | 2.62 | 92.86 | 7.14 | 9.76 | 30 | 118 |

**Summary.** AlphaTrans can successfully transform projects to remove method and constructor overloading. Moreover, it can automatically translate 91.99% of the source PL types and use that to create and validate project skeletons in the target PL.

**Effectiveness in Compositional Translation and Validation.** Table 8 shows the compositional translation and validation results. The *AMF* column indicates the total number of application method fragments. The numbers in subsequent columns demonstrate the effectiveness of AlphaTrans in the translation and validation of *AMF*s only.[7] The *Syntax Check* column indicates the percentage of *AMF*s that pass syntactic validation. Column *SNEF* shows the percentage of *AMF*s not covered by source project tests. AlphaTrans successfully generates syntactically correct code (98.80% of AMFs and 96.40% of all fragments—field, application, and test fragments—across ten subjects). We also observe that 43.43% of *AMF*s are not covered during the execution of any test, i.e., we cannot go beyond syntactic check and validate their runtime behavior or functional equivalence.

[7]Please refer to our artifact [41] for details of all the translation and validation of other fragments shown in Table 7.

For 56.57% of the *AMF*s that are covered by source project tests, AlphaTrans attempts to validate their functional equivalence using GraalVM. Multi-column *GraalVM* shows the percentage of *AMFs* that GraalVM executes and successfully validates (*GS*), executes but there is a test assertion failure (*GF*), and cannot execute due to its limitation (*GE*) mentioned in §5.6.1. 24.50% (min=6.01% and max=70.70%), 16.23% (min=1.04% and max=32.50%), and 15.84% (min=3.13% and max=25.11%) of *AMFs* resulted in *Graal Success*, *Graal Fail*, and *Graal Error*, respectively. Note that these numbers add up to 56.57% of *AMFs* that were covered by source project tests. With respect to only covered methods, GraalVM Success is 43.30%. Furthermore, our analysis shows that a high portion of methods that are not covered by tests are either abstract methods or getter/setter methods. If their translations are syntactically correct, they are also likely functionally equivalent, which can ramp up the success rate (§5.7.5). Spearman Rank Order Correlation [196] indicates a strong positive correlation between method coverage (Table 7) and *GS* numbers ($\rho = 0.92$), confirming that with better method coverage, validated *AMF*s are very likely to be higher.

Regardless of GraalVM's validation for functional equivalence, AlphaTrans translates and executes the test fragments on the recomposed translated project. The *Test Translation* multi-column shows the results of test translation and execution. Column *TNEF* indicates the ratio of *AMF*s where execution of translated tests never reached them. Columns *ATP* through *ATF* show the number of *AMFs* that AlphaTrans executed using translated test fragments, categorized per the test execution results. For 2.88% of the *AMFs*, all the test fragments that covered them were marked as pass (*ATP*). For 3.72% and 5.44%, at least one test (*OTF*) or more than one test (*MTF*) failed. All the test fragments failed for 2.62% of the *AMFs*. Note that these numbers (TNEF, ATP, OTF, MTF, and ATF) add up to 56.57% of the AMFs that are covered by source project tests.

For cases with test failure, columns *RE* and *AF* show the breakdown of whether test failure was due to assertion failure or runtime error. We can observe that most test executions terminated with a runtime error due to translation bugs and never reached an assert statement. *Our manual investigation confirms that a high rate of runtime errors is due to a relatively small number of fragments with translation bugs.* Although test decomposition helps with *test translation coupling effect* (§5.2), there is still a high degree of runtime errors due to long call chains in these projects (the average number of methods executed per test in the original and decomposed test suites are 27.4 and 21.8, respectively). As a result, the overall pass rate, i.e., the percentage of recomposed test fragments for the translated projects that pass (column *TPR*), is low. These results also confirm the necessity of in-isolation testing through language interoperability by AlphaTrans.

We also calculated the number of *AMFs* that GraalVM could not execute (numbers under *GE* column) but translated test fragments exercised (column *M1*). *All* indicates the number of *AMF*s with all passing tests, including test fragments with assert statements, indicating the validation of functional equivalence (with respect to the source project tests). *Some* corresponds to the number of *AMF*s with at least one passing test, which indicates runtime validation. Overall, test translation validates the functional correctness and runtime behavior of 30 and 118 fragments that GraalVM could not exercise.

Finally, we analyzed translations using PyLint [197], which scores Python files on a scale of 0 to 10 based on how Pythonic the code is. All AlphaTrans translations achieved scores of 10, mainly because (1) LLMs inherently generate idiomatic code and (2) AlphaTrans uses Black [198] for formatting the translations extracted from the LLM response. These results confirm that the translations are all Pythonic, i.e., they adhere to Python coding standards and best practices.

**Summary.** AlphaTrans effectively performs compositional translation and validation of 17,874 fragments, achieving overall 96.40% syntactic correctness (98.80% for AMFs), 27.03% runtime behavior validation (GS+M1 Some), and 25.14% functional equivalence (GS+M1 All).

### 5.7.3 RQ2: Translation Bugs and Fixes

We investigated the manual effort for fixing translation bugs in a subset of studied subjects, namely *Commons-FileUpload*, *Commons-CLI*, *Commons-CSV*, and *Commons-Validator*. We also discuss some of the translation bugs and fixes for them to better illustrate the challenges in code translation.

**Human Study.** Our two human subjects were selected due to their relative familiarity with the selected projects. Their effort indicates an upper bound for the amount of time required to fix translation bugs since developers of the source projects are likely to fix the bugs better and faster. We shared with them the source program in Java, the translations by AlphaTrans, and all the reports and artifacts generated by AlphaTrans during translation.

For *Commons-FileUpload*, achieving green tests took 5.5 hours and required 120 and 114 line additions and deletions from partial translations. For *Commons-CLI*, the manual fix took 11 hours, making 614 and 1,253 line additions and deletions, respectively. The project was very dense for *Commons-CSV*, with many method calls, making it harder to manually fix bugs. Nevertheless, a developer achieved all green tests in 30 hours with 2,676 and 999 line additions and deletions, respectively. Finally, for *Commons-Validator*, the developer spent 34 hours to fix translation bugs, with 3,585 and 2,416 line additions and deletions, respectively. One of the major feedback from developers was that test decomposition greatly helped locate and fix translation bugs: in case of a test failure, developers only need to investigate the last call statement in the failed test fragment instead of looking at the stack trace and other prior calls.

**Translation Bugs.** Our artifacts [41] contain partial translations and fixed versions as separate commits. These commits can serve as useful benchmarks for evaluating fault localization, program repair, and test generation techniques. This section shows *four* instances of such translation bugs.

The two most prevalent sources of translation bugs are mismatches between APIs and behavioral differences in the PLs. The code snippet below demonstrates a bug that happened due to a mismatch in the logic of `Calendar` (Java) and `datetime` (Python). Line 3 in Java sets the `MONTH` field to `0`, which corresponds to the first month of the year (January). Similarly, the Python translation sets the `month` attribute to `0`; however, in the Python library, January corresponds to value 1 for `month`.

```
1 ------------------ JAVA SOURCE CODE -----------------
2 Calendar calendar = Calendar.getInstance()
3 calendar.set(Calendar.MONTH, 0);
4
```

```
1 ----------------- PYTHON TRANSLATION ----------------
2   calendar = datetime.datetime.now()
3 - calendar = calendar.replace(month=0)
4 + calendar = calendar.replace(month=1)
```

The next example shows the difference in implicit type casting between the two Pls. Line 5 in Java source code concatenates a `String` value with `null`. During execution, Java runtime silently casts `null` to a `String` and then performs the concatenation operation on it. In Python, concatenating an `str` with `None` results in a `TypeError` as the operands of the binary operation has different types. A correct Python translation requires explicit casting of `None` to `str` as shown in Line 5.

```
1 ------------------ JAVA SOURCE CODE -----------------
2 qChar = "'";
3 nullStr = null;
4
5 this.qNullStr = qChar + nullStr + qChar;
```

```
1 ----------------- PYTHON TRANSLATION ----------------
2   qChar = "'"
3   nullStr = None
4 - self.qNullStr = qChar + nullStr + qChar
5 + self.qNullStr = qChar + str(nullStr) + qChar
```

The third example shows an instance of `write(int b)` method from `ByteArrayOutputStream` class, where the least significant 8 bits of the integer `(b2 << 4) | (b3 >> 2)` are directly written to the stream. The incorrect Python translation attempts to construct a `bytes` object using a singleton list with the input integer before writing it to an object of type `io.BytesIO`. However, this neglects that the `bytes()` constructor requires the integers in the input iterable to be strictly in the range of `[0, 255]`. Thereby, a `ValueError` is thrown when `b2` is large. The correct translation requires `0xF`, a mask that maintains only the 4 lowest bits of `b2` before left-shifting by 4 as shown in Line 3 under Python translation. Given that `b3` and `b4` each contain no more than 8 bits, this change ensures the least significant 8 bits of `(b2 << 4) | (b3 >> 2)` are correctly written to the `BytesIO` object.

```
------------------ JAVA SOURCE CODE -----------------

out.write((b2 << 4) | (b3 >> 2));
```

```
----------------- PYTHON TRANSLATION ----------------
- out.write(bytes([(b2 << 4) | (b3 >> 2)]))
+ out.write(bytes([((b2 & 0xF) << 4) | (b3 >> 2)]))
```

The last code snippet demonstrates the Java behavior of an iterator that is unavailable in Python. The incorrect Python translation uses `next()` to implement both `next()` and `hasNext()` methods of the Java `java.util.Iterator` type. However, calling `next()` increments the iterator in Python. The correct translation should implement `PeekableIterator` interface in Python with a method `hasNext() -> bool`.

```
------------------ JAVA SOURCE CODE -----------------
Iterator<String> headers = ls.keySet().iterator();

assertEquals("content", headers.next());

assertFalse(headers.hasNext());
```

```
----------------- PYTHON TRANSLATION ----------------
- headers = iter(ls.keys())
+ headers = PeekableIterator(ls.keys())
  self.assertEqual("content", next(headers))
- self.assertFalse(next(headers, None) is not None)
+ self.assertFalse(headers.hasNext())
```

*Implications.* To obtain correct translation, especially for code using library APIs, models need to generate test cases as well that can validate the translated fragment in isolation. This could be an interesting direction for applying an agentic approach, where the orchestrating agent can decide when to generate test cases, and the test case generator agent gets all the information by running static analysis tools, gathering context from previous runs, collecting API documentation by crawling the internet, and finally generating the translation based on all the information.

**Summary.** Although AlphaTrans cannot validate all the translations, it provides partial translations and artifacts that developers can use to complete the translation and achieve green tests in a reasonable time (20.1 hours, on average).

### 5.7.4 RQ3: Impact of Test Decomposition

We previously showed the effectiveness of test translation in validating the runtime behavior or functional correctness of translated method fragments (§5.7.2). To better understand how test decomposition helps address *test translation coupling effect*, we collected translated unit tests with at least two decomposed test fragments. We further applied filtering and kept only those tests for which *all* their decomposed fragments were

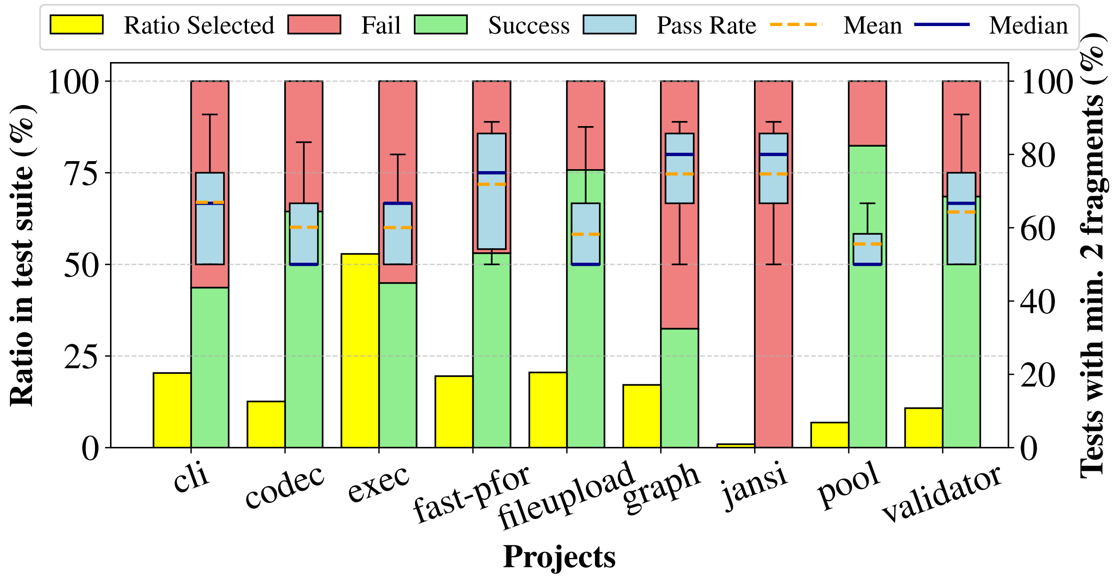



Figure 13: Effectiveness of test decomposition in AlphaTrans for validating earlier fragments in failing tests.

executed, regardless of passing or failing outcomes. The yellow bars in Figure 13 show the percentage of selected unit tests from the translated test suites. None of the tests met the criteria for *Commons-CSV*, so we excluded it from further investigation.

We categorized the selected unit tests into two groups: those with all passing test fragments (green bars in Figure 13) and those with at least one failing test fragment (red bars in Figure 13). For the unit tests in the latter group, we calculated the pass rate of the *decomposed test fragments*. The blue box chart in Figure 13 shows the distribution of the measured pass rate per unit test. These unit tests would have been marked as *fail* without test decomposition. However, we can observe that these tests can be decomposed into test fragments such that 62.41% of them pass. These results indicate how decomposed test fragments were useful in helping developers localize translation bugs more easily and resolve translation bugs faster.

Table 9: Effectiveness of test augmentation in exercising and validating more application method fragments. Abbreviations in the table stand for **ATP**: Fragments All Test Pass and **TPR**: Test Pass Rate. ATP+ and TPR+ demonstrate ATP and TPR gain through test augmentation.

| Subjects | Developer-Written Test | | | | | EvoSuite Test | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Method Coverage (%) | # Decomposed Tests | Avg. Methods Executed / Test | TPR (%) | ATP (%) | Method Coverage (%) | # Tests | Avg. Methods Executed / Test | TPR+ (%) | ATP+ (%) |
| cli | 94.14 | 3036 | 34.25 | 10.08 | 8.42 | 95.97 | 569 | 12.15 | 2.99 | 1.47 |
| codec | 91.03 | 3522 | 10.56 | 9.43 | 4.12 | 80.74 | 1141 | 8.02 | 3.51 | 0.88 |
| csv | 90.64 | 1219 | 52.62 | 0 | 0 | 74.04 | 220 | 39.16 | 0 | 0.00 |
| exec | 54.84 | 311 | 18.99 | 19.29 | 4.44 | 61.29 | 245 | 6.32 | 3.27 | 1.21 |
| fast-pfor | 54.55 | 249 | 41.62 | 20.08 | 4.28 | 39.17 | 1843 | 4.31 | 5.59 | 1.07 |
| fileupload | 13.02 | 93 | 3.54 | 63.44 | 3.65 | 70.31 | 231 | 5.29 | 11.26 | 2.60 |
| graph | 58.78 | 933 | 25.02 | 11.04 | 0.00 | 76.71 | 800 | 9.00 | 5.13 | 0.92 |
| jansi | 23.47 | 187 | 13.57 | 1.07 | 0.24 | 51.83 | 332 | 9.08 | 3.31 | 0.73 |
| pool | 22.29 | 287 | 6.52 | 6.62 | 1.61 | 37.24 | 394 | 7.36 | 10.41 | 2.20 |
| validator | 63.31 | 1479 | 11.68 | 11.70 | 3.25 | 81.42 | 1305 | 13.43 | 9.73 | 7.59 |
| **Total** | 56.57 | 11316 | 21.84 | 9.76 | 2.88 | 66.87 | 7080 | 11.41 | 5.85 | 2.11 |

**Summary.** Test decomposition unburdens validation of translated method fragments from incorrect translations. 62.41% of test fragments for unit tests that would have been marked as failed achieved passing outcomes.

### 5.7.5 RQ4: Impact of Test Coverage

Although existing developer-written tests are useful for checking functional equivalence, they can pose two major issues for automated code translation and validation. First, the coverage for these tests can be extremely low (e.g., 13.02% for Commons-FileUpload [191] as reported in Table 7), preventing most of the code from being validated; as our investigation of RQ1 showed, the translation validation rate strongly correlates with test suites' (method) coverage. Second, a developer-written test can have a long call sequence. To show the positive impact of more-focused tests with higher coverage on translation validation, we automatically generated additional tests using EvoSuite [199]. We used default tool configuration with a time budget of 120 seconds per class.

Table 9 compares properties of the developer-written and EvoSuite-generated tests. Evosuite tests cover more methods than the developer-written test suite (66.87% *Method Coverage* compared to 56.57%). Furthermore, the average number of methods executed per single test is almost half that of decomposed test suites (11.41 compared to 21.84

methods). To demonstrate the impact of test quality on AlphaTrans's overall performance, we translated and executed the EvoSuite-generated tests. Corroborated by the numbers under *TPR+* and *ATP+* columns, we can see that augmenting the test suite increases the method coverage, and thereby, the TPR and ATP numbers from RQ1 by 5.85% and 2.11%, respectively. Not all EvoSuite tests have assertions, and even if they do, the quality of the assertions could be lower compared to developer-written tests (e.g., checking trivial properties with weak fault detection ability). Nonetheless, the higher *TPR* and *ATP* of such tests enhance runtime validation, which is still promising in code translation. EvoSuite is incompatible with Java 21, and hence GraalVM, which prevented us from using Evosuite-generated tests in AlphaTrans. We anticipate that incorporating it in AlphaTrans could improve the overall quality of translations.

**Summary.** Augmenting the existing test suite increases code coverage, thereby exercising and validating more AMFs. Test augmentation can further validate the correctness of 2.11% of fragments not executed by developer tests. The generated tests are more focused and, on average, invoke 48% fewer methods than the developer-written tests.

### 5.7.6 RQ5: Ablation Study

We performed three ablation studies to investigate the impact of program transformation, choice of LLM, and program decomposition on the performance of AlphaTrans.

**Impact of Program Transformation.** We removed the program transformation component of AlphaTrans and executed the entire pipeline. The results in Table 10 show that without transformation (e.g., resolving method/constructor overloading), the performance of AlphaTrans drops significantly: *GS*, *ATP*, and *TPR* values decreased to 4.58% (from 24.50%), 0.81% (from 2.88%), and 3.05% (from 9.76%), respectively. This is because Python does not support overloading and only considers the last method/constructor implementation, resulting in runtime errors or test failures. Also,

Table 10: Importance of program transformation. Abbreviations in the table stand for **AMF**: #Application Method Fragments, **SNEF**: Source Non-Exercised Fragments, **GS**: Graal Success, **GF**: Graal Fail, **GE**: Graal Error, **TNEF**: Target Non-Exercised Fragments, **ATP**: Fragments All Test Pass, **OTF**: Fragments One Test Fail, **MTF**: Fragments Many Test Fail, **ATF**: Fragments All Test Fail, and **TPR**: Test Pass Rate.

| **Subjects** | **AMF** | **SNEF** (%) | **GraalVM** | | | **Test Translation** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | **GS** (%) | **GF** (%) | **GE** (%) | **TNEF** (%) | **ATP** (%) | **OTF** (%) | **MTF** (%) | **ATF** (%) | **TPR** (%) |
| cli | 276 | 5.80 | 0 | 0 | 94.20 | 85.51 | 0 | 2.17 | 1.09 | 5.43 | 0.46 |
| codec | 678 | 10.62 | 0 | 0 | 89.38 | 72.86 | 2.06 | 5.46 | 2.80 | 6.19 | 5.04 |
| csv | 235 | 8.51 | 0 | 0 | 91.49 | 88.09 | 0.43 | 0.43 | 0.85 | 1.70 | 0.32 |
| exec | 253 | 46.25 | 26.88 | 4.35 | 22.53 | 49.41 | 1.58 | 1.19 | 0 | 1.58 | 4.29 |
| fast-pfor | 754 | 46.02 | 0 | 0 | 53.98 | 38.20 | 0.93 | 6.63 | 0.80 | 7.43 | 4.88 |
| fileupload | 168 | 85.12 | 10.71 | 0.60 | 3.57 | 8.93 | 1.19 | 2.98 | 1.19 | 0.60 | 23.08 |
| graph | 556 | 40.65 | 0 | 0 | 59.35 | 56.12 | 0 | 1.98 | 0.36 | 0.90 | 5.48 |
| jansi | 314 | 69.75 | 0 | 0 | 30.25 | 29.94 | 0 | 0.32 | 0 | 0 | 0 |
| pool | 574 | 70.56 | 0 | 0 | 29.44 | 27.00 | 1.39 | 0 | 0.35 | 0.70 | 5.48 |
| validator | 623 | 33.39 | 18.78 | 20.06 | 27.77 | 66.29 | 0 | 0 | 0.32 | 0 | 0.43 |
| **Total** | 4431 | 40.01 | 4.58 | 3.09 | 52.31 | 52.79 | 0.81 | 2.57 | 0.86 | 2.96 | 3.05 |

*GE* values increase due to the interference of overloaded code constructs with GraalVM.

**Choice of LLM.** For this experiment, we replaced the DeepSeek-Coder-33b-Instruct with GPT-4o and repeated the entire pipeline of AlphaTrans (Table 11). A stronger model such as GPT-4o improves the translation quality—functional equivalence increases from 25.14% to 27.95%. For some projects, the ATP and TPR rates are higher for DeepSeek-Coder-33b-Instruct translations, whereas for the others, GPT-4o results in higher values. We investigated each LLM's successful AMF translations to better understand the differences. We observed a huge overlap between successful AMFs and the unique benefits each LLM provides in code translation (Figure 14). GPT-4o handles API translation and type casting better, resolving the first three translation bugs discussed in §5.7.3. In contrast, it tends to add unnecessary code, mostly due to error handling, which results in a functional mismatch. In the example below, `create2` method can take a `None` value, and its implementation performs error handling when necessary. GPT-4o adds unnecessary error handling code, interfering with program logic and resulting in test failures.

Table 11: Effectiveness of AlphaTrans with GPT-4o in compositional translation and validation. Abbreviations in the table stand for **AMF**: #Application Method Fragments, **SNEF**: Source Non-Exercised Fragments, **GS**: Graal Success, **GF**: Graal Fail, **GE**: Graal Error, **TNEF**: Target Non-Exercised Fragments, **ATP**: Fragments All Test Pass, **OTF**: Fragments One Test Fail, **MTF**: Fragments Many Test Fail, **ATF**: Fragments All Test Fail, **TPR**: Test Pass Rate, **O**: Overall, **RE**: Runtime Error, **AF**: Assertion Failure, and **M1**: Number of *AMFs* that GraalVM could not execute (*GE*) but translated test fragments exercised.

| Subjects | Syntax Check (%) | SNEF (%) | GraalVM | | | Test Translation | | | | | | | | | | | M1 | | Cost ($) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | GS (%) | GF (%) | GE (%) | TNEF (%) | ATP (%) | OTF (%) | | | MTF (%) | | | ATF (%) | | | TPR (%) | | | |
| | | | | | | | | O | RE | AF | O | RE | AF | O | RE | AF | | All | Some | |
| cli | 99.63 | 5.86 | 76.92 | 8.79 | 8.42 | 78.39 | 4.76 | 3.66 | 90.00 | 10.00 | 6.23 | 100 | 0 | 1.10 | 100 | 0 | 2.47 | 0 | 1 | 19.69 |
| codec | 97.79 | 8.97 | 42.50 | 28.53 | 20.00 | 74.56 | 3.82 | 4.12 | 78.57 | 21.43 | 6.18 | 30.62 | 69.38 | 2.35 | 73.33 | 26.67 | 9.57 | 2 | 18 | 39.97 |
| csv | 98.30 | 9.36 | 41.28 | 25.96 | 23.40 | 86.81 | 0 | 0 | 0 | 0 | 1.70 | 78.17 | 21.83 | 2.13 | 96.72 | 3.28 | 0.98 | 0 | 8 | 16.66 |
| exec | 99.60 | 45.16 | 35.89 | 2.42 | 16.53 | 44.76 | 4.84 | 0 | 0 | 0 | 5.24 | 0 | 100 | 0 | 0 | 0 | 28.62 | 1 | 1 | 3.90 |
| fast-pfor | 97.46 | 45.45 | 14.71 | 16.98 | 22.86 | 52.94 | 1.47 | 0 | 0 | 0 | 0 | 0 | 0 | 0.13 | 100 | 0 | 2.41 | 2 | 0 | 12.93 |
| fileupload | 99.48 | 86.98 | 9.90 | 0.52 | 2.60 | 4.17 | 6.77 | 1.56 | 0 | 100 | 0.52 | 100 | 0 | 0 | 0 | 0 | 54.84 | 1 | 3 | 2.25 |
| graph | 98.71 | 41.22 | 27.73 | 19.04 | 12.01 | 58.78 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 12.03 |
| jansi | 99.76 | 76.53 | 8.07 | 11.98 | 3.42 | 23.47 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3.66 |
| pool | 99.71 | 77.71 | 7.48 | 1.32 | 13.49 | 21.70 | 0.59 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2.09 | 0 | 0 | 7.33 |
| validator | 99.23 | 36.69 | 38.24 | 15.94 | 9.13 | 54.95 | 4.02 | 2.32 | 93.33 | 6.67 | 1.55 | 70.59 | 29.41 | 0.46 | 100 | 0 | 10.41 | 0 | 3 | 25.53 |
| **Total** | 98.80 | 43.43 | 27.83 | 14.54 | 14.20 | 50.64 | 2.26 | 1.20 | 80.36 | 19.64 | 1.87 | 59.39 | 40.61 | 0.60 | 86.59 | 13.41 | 6.45 | 6 | 34 | 143.95 |

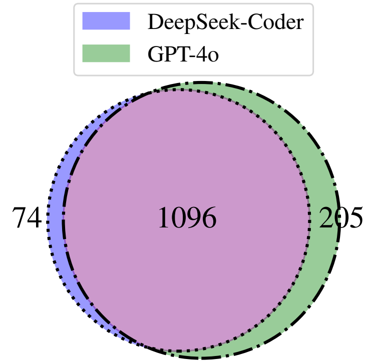
```java
------------------ JAVA SOURCE CODE -----------------
public Option create1(OptionGroup optGroup) {
    return create2(optGroup);

}
```

```python
----------------- PYTHON TRANSLATION ----------------
def create1(self, optGroup: OptionGroup) -> Option:
+   if optGroup is None:
+       raise ValueError("optGroup is None")
    return self.create2(optGroup)
```

It is worth noting that using commercial LLMs comes at a cost. The last column of Table 11 (column *Cost*) shows the cost of using GPT-4o for repeating the experiments, resulting in the total cost of \$143.95 for translating all the subjects (average cost of \$14.39 per project).

**Impact of Program Decomposition.** For this ablation study, we prompted GPT-4o and DeepSeek-Coder-33b-Instruct *file-by-file* and evaluated translation correctness through test execution and GraalVM (Table 12). Not surprisingly, a considerable percentage of the files exceeded the model context window size, particularly for DeepSeekCoder, with 9.08% of the files encountering this problem. Among the prompted files, 21.36% (19.44% for DeepSeekCoder and 1.92% for GPT-4o) were syntactically incorrect. Translations that passed the syntactic correctness check did not pass any translated test execution, whereas GraalVM validated 2.56% of the files in total (1.02% for DeepSeekCoder and 1.54% for GPT-4o). Note that file-level GraalVM validation does not mean that all the methods are correctly translated and validated—only the methods in the class that are executed by tests. Manual analysis of

Figure 14: Functional equivalence overlap of AMFs (Graal Success) between LLMs.




the results of this experiment revealed that one prominent culprit of test failures was LLM hallucination with method/variable names. Given that we had no skeleton construction in this baseline, such issues could not be avoided; this demonstrates the usefulness of skeleton construction and incremental translation in AlphaTrans.

**Summary.** Omitting program transformation and program decomposition significantly lowers the effectiveness of AlphaTrans. A stronger model such as GPT-4o resolves non-trivial issues concerning type casting and API translation but may result in trivial translation bugs. When possible, users of AlphaTrans can prompt multiple LLMs to achieve better translation performance.

### 5.8 Related Work

There are generally two main categories of techniques for translating code from one PL to another: (1) using transpilers and statistical machine translation and (2) leveraging language models.

#### 5.8.1 Code Translation Using Non-LLM-Based Approaches

Tools like C2Rust [18], CxGo [19], Sharpen [20], and Java2CSharp [21] translate code from C to Rust, C to Go, and Java to C# respectively. A series of statistical machine

Table 12: Importance of program decomposition. Abbreviations stand for **GS**: Graal Success and **TPR**: Test Pass Rate. Since Java can contain multiple classes in one file, **#Files** is smaller from **#Classes** in Table 7.

| **Subjects** | **# Files** | **GPT-4o** | | | | **DeepSeek-Coder** | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **Over Context** | **Syntax Error** | **GS** | **TPR (%)** | **Over Context** | **Syntax Error** | **GS** | **TPR (%)** |
| cli | 51 | 0 | 0 | 1 | 0 | 5 | 6 | 0 | 0 |
| codec | 136 | 0 | 1 | 7 | 0 | 28 | 46 | 4 | 0 |
| csv | 33 | 0 | 3 | 0 | 0 | 6 | 8 | 0 | 0 |
| exec | 54 | 0 | 0 | 0 | 0 | 1 | 4 | 0 | 0 |
| fast-pfor | 85 | 1 | 1 | 1 | 0 | 12 | 30 | 1 | 0 |
| fileupload | 43 | 0 | 0 | 0 | 0 | 2 | 4 | 0 | 0 |
| graph | 159 | 0 | 6 | 3 | 0 | 0 | 17 | 3 | 0 |
| jansi | 30 | 0 | 0 | 0 | 0 | 3 | 10 | 0 | 0 |
| pool | 72 | 0 | 1 | 0 | 0 | 4 | 8 | 0 | 0 |
| validator | 119 | 0 | 3 | 0 | 0 | 10 | 19 | 0 | 0 |
| **Total** | 782 | 1 | 15 | 12 | 0 | 71 | 152 | 8 | 0 |

translation techniques [24], [25], [26], [27] focus on translating Java to C#. Deep learning approaches have also been applied for code translation [29], [30]. None of these efforts have tackled translating real-world Java projects to Python. LLM-based techniques are also superior to transpilers in terms of performance or readability [22].

### 5.8.2 Code Translation Using LLMs

Recently, LLMs have been employed for code translation [22], [56], [57], [58], [59], [60], demonstrating high success rates on crafted examples but poor performance on real-world projects. Other studies [64], [128] have also applied language models for code translation, mainly focusing on crafted benchmarks. Concurrently to our work, two other techniques for repository-level code translation focusing on different language pairs were proposed [75], [76]. Syzygy [75] translates repository-level C to Rust using GPT-4. Oxidizer [76] leverages language feature mapping and type-driven techniques for translating Go to Rust. Both techniques use I/O equivalence for validating their translations. There are also approaches that use transpiler output to guide LLM-based code translation [85]. However, the limitation of such work is the availability of robust and well-maintained transpilers, which, in many cases, may not be feasible. Nitin et al. [84] presented a specification-based translation, where a natural language specification is captured from the source code, which helps the translation process. Yang et al. [86] used tests to assist the translation. Compared to previous work, our contributions include a compositional and validation-guided code translation approach that leverages two types of validation techniques and evaluation of the approach on 10 real-world projects.

### 5.9 Threats to Validity

**External Validity.** One of the key external threats is the generalizability of AlphaTrans. We built and evaluated the first version of AlphaTrans to translate from Java to Python. However, our pipeline is generic, and with minimal effort, the current implementation can translate Java programs to more target languages (e.g., languages supported by GraalVM). Furthermore, the majority of the tools that we used support a large set of programming languages such as JavaScript, Ruby, C/C++, and Rust.

**Internal Validity.** One threat can be the manual validation of the translated types. To address that, several authors verified the types individually and consulted API documents when necessary. Another threat is that, while all successes reported by the GraalVM validation are true successes, we may have *underestimated* the capabilities of AlphaTrans by a considerable margin due to the significant proportion of errors caused by limitations in the GraalVM validation approach. To mitigate this threat, we manually augmented the universal type map to support a more comprehensive translation of types. That said, we have implemented *driver* code templates to provide a mechanism for adding support for more types by the users of AlphaTrans if needed.

**Construct Validity.** In order to mitigate construct validity, AlphaTrans is built and validated with well-vetted tools, such as GraalVM [91], JaCoCo [184], Python coverage [185], CodeQL [181], etc.

### 5.10 Concluding Remarks

In this paper, we introduced AlphaTrans, a neuro-symbolic approach that combines the power of static analysis and abilities of LLMs in code synthesis to automate repository-level code translation and validation. AlphaTrans decomposes the program into smaller fragments and translates the fragments in reverse call order, incrementally building the source project in the target language. In addition to syntactic checks, AlphaTrans implements two types of validation through GraalVM and test translation. AlphaTrans is the first approach to translate and validate real-world projects, and we envision several research directions to advance repository-level code translation and validation.

One of the major challenges in repository-level code translation is identifying suitable library APIs in the target PL. Often, equivalent Python APIs may not exist, requiring new code generation or translation of the library API itself. Even if similar libraries exist, the logic of libraries might be different in two PLs. AlphaTrans supports translating frequently used APIs and aims to build a generic pipeline. Supporting all the libraries in the pipeline remains an open challenge that we aim to address in future work. Furthermore, while the idea of compositional translation and validation is PL-agnostic, the static analysis makes the extension of AlphaTrans to translating from other source projects challenging. Devising LLM-enabled or PL-agnostic static analysis approaches can benefit code translation approaches such as AlphaTrans. We also showed that the quality of the source project test suite can significantly impact the translation validation results. In future work, we plan to integrate an LLM-based test generator into the AlphaTrans pipeline to enhance the validation component.

# Chapter 6 MatchFixAgent: Language-Agnostic Autonomous Repository-Level Code Translation Validation and Repair

## 6.1 Introduction

Code translation, the process of converting source code from one programming language (PL) to another, is a cornerstone of software modernization efforts that enhance performance, maintainability, and reliability [1], [2], [3], [170]. Translation validation and repair are integral steps in code translation for determining functional equivalence and patch generation for incorrect translations. However, performing validation and repair manually—particularly in large codebases—can be tedious, time-consuming, and error-prone, especially when complex code structures and dependencies are involved [200], [201], [202]. Prior work defines value and type equivalences and translations to compare source and target implementations over pairs of concrete inputs. The inputs either come from existing tests from the source project [45], [75], [76], [77] or differential fuzzing [85], [94]. Despite notable advancements in validation and repair of repository-level code translation, existing techniques are hampered by the following limitations (examples given in §6.2):

1. *Difficulty Generalizing to Many PL Pairs.* While the fundamental ideas of current validation approaches extend to many PL pairs, their actual implementations typically support just one language pair. This is because supporting language interoperability between a pair of PLs requires a large engineering effort, as evidenced by the size of these tools[8]. Given the quadratic number of PL pairs, language interoperability techniques are extremely challenging to scale to many PL pairs.

2. *Unknown Test Requirements.* Current translation validation approaches require a

[8] The implementation of notable recent translation and validation techniques are AlphaTrans (10859 LoC), Oxidizer (19052 LoC), and Skel (3843 LoC).

set of valid inputs to validate the input-output equivalence between source functions and their translation. They generate these inputs by either executing available source tests [45], [75], [76], [77] or fuzzing the source project [85], [94] Unit tests are often incomplete, missing important inputs, and resulting in false claims of equivalence. Fuzzing techniques suffer from generating invalid inputs, also resulting in false claims of inequivalence, and in general, failing to reach deep into the code or create complex objects in the context of real-world projects [93], [95].

3. *Ineffective Translation Repair.* Recent studies have shown that a more rigorous validation can reveal more translation bugs [22]. Hence, advancement in translation validation should be accompanied by effective repair strategies. Existing techniques, however, either require the user to fix incorrect translations manually [77] or use feedback-driven re-prompting strategies [45], [76] that are barely effective in repository-level code translation due to long call-chain dependencies in real-world projects [45].

Prior work consists of high-effort implementations that can deliver inconsistent results and only apply to one of a quadratic number of language pairs. Large Language Models (LLMs) have recently been successful at same-language equivalence validation [203], [204], so replacing cross-language equivalence implementations with LLM decisions is a logical next step. While there is a risk of incorrect results, the baseline mechanical approaches already display low accuracy.

Towards this end, we propose MatchFixAgent, a *language-agnostic* approach to automate *validation* and *repair* of repository-level code translation (§6.3.1). MatchFixAgent combines the generative power of LLMs with several approximate code semantic analyses, i.e., analysis of control- and data-flow paths, library APIs, exception handling, and (formal or informal) specification (§6.3.2). These semantic analyses are then fed to a *test generator & repair agent* to generate tests for assessing functional equivalence, and repair the translation in the case of failing tests (§6.3.3). The final equivalence decision will be made by the *verdict* agent, considering the approximate semantic analyses and test execution results (§6.3.4). MatchFixAgent is very lightweight (1,650 lines of code), modular, and interoperable with existing repository-level translation systems.

We evaluate the effectiveness of MatchFixAgent for repository-level translation validation and repair against four existing techniques [45], [76], [77], [88]. Our benchmark comprises 2,219 source–translation function pairs, which cover 6 PL pairs, drawn from 24 real-world projects totaling over $900K$ lines of code (details in Appendix 6.4.1). For each translation pair, we obtain an equivalence verdict (validation outcome) from both MatchFixAgent and other techniques. Overall, MatchFixAgent returns a verdict for 99.2% of pairs, while alternative approaches do so for only 71.6% (§6.4.2). On the 1,571 pairs where both produce verdicts, MatchFixAgent agrees with other approaches in 72.8% of cases. For the remaining disagreements, a systematic manual investigation finds MatchFixAgent to be correct in 60.7% of cases and incorrect in the rest. In translation repair, MatchFixAgent can fix 50.6% of translation bugs, 32.1% more than existing approaches (§6.4.3). We show that MatchFixAgent is compatible with different LLMs and agent frameworks, producing comparable results (§6.4.4). Lastly, our ablation study shows that removing code analyses and in-the-loop test generation reduces verdict accuracy by 42.3%, while increasing token usage by 5.2% (§6.4.5). These results confirm that MatchFixAgent is a viable alternative to prior work's validation and repair approaches, while being vastly easier to adapt to new PL pairs. Our notable contributions are:

1. We present MatchFixAgent, a PL-agnostic, agentic approach for validation and repair of repository-level code translation.

2. We demonstrate that MatchFixAgent is a viable alternative to prior work's validation approach, while being vastly easier to adapt to new PL pairs.

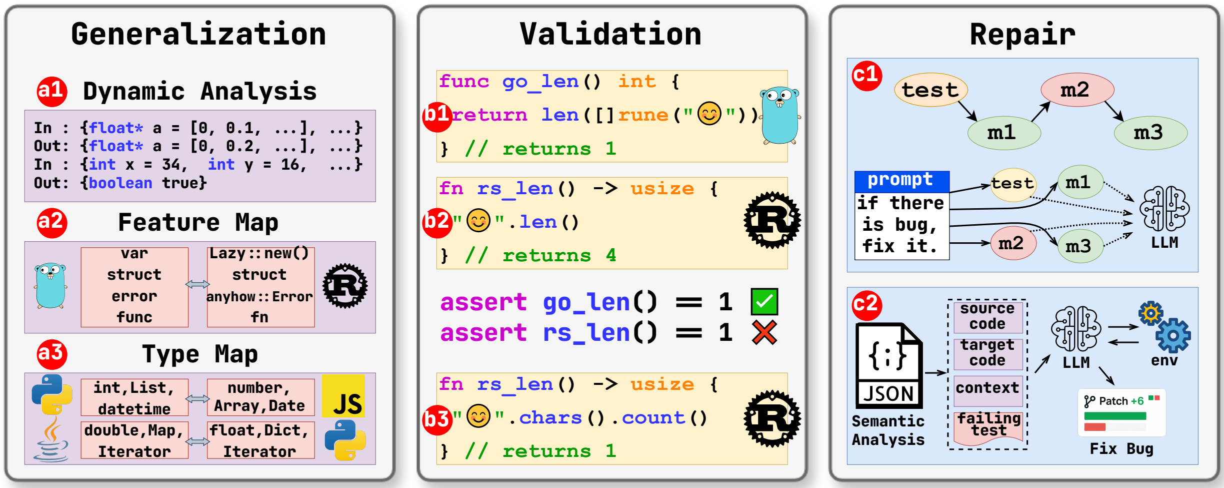



Figure 15: Illustration of key limitations of existing techniques in validation and repair of repository-level code translation and MatchFixAgent addressing them.

3. We show the benefit of MatchFixAgent's multi-agent architecture compared to standalone-agent design.

### 6.2 Limitations of Prior Work

To demonstrate the limitations of existing techniques [45], [75], [76], [77] for validation and repair, we use the examples in Figure 15.

**Limitation 1: Generalization.** Existing automated translation tools require substantial engineering effort to build validation systems for individual language pairs. For instance, Oxidizer [76] and Syzygy [75] require dynamic analysis and I/O extraction from source code ( a1 ). Oxidizer further requires a predefined feature map ( a2 ), which together with their dynamic analyzer is 19,000 lines of code. Other tools like AlphaTrans [45] and Skel [77] validate programs using high quality type map ( a3 ), which requires manual maintenance over time as PLs evolve. In comparison, MatchFixAgent uses only language-agnostic LLM prompts, an off-the-shelf LLM coding agent tool, and lightweight static analysis. While the static analysis must be implemented for each PL, each PL requires approximately 280 additional lines of code to support, hence it is extremely easy to generalize to many PLs.

**Limitation 2: Validation.** Most prior work depends on existing source project tests for translation validation, but these tests may have insufficient coverage. The test suites in AlphaTrans [45] have an average of 56.57% method coverage. Even when developer-written tests provide adequate coverage, they may miss the edge cases that expose subtle semantic differences between source and target languages. Consider the real-world case from Oxidizer [76], where a `Go` program ( b1 ) that counts characters in a string is translated to `Rust` ( b2 ) using the `.len()` method, which counts bytes rather than characters. Oxidizer validates this pair as *functionally equivalent* because it only exercises this program with ASCII inputs. MatchFixAgent, in contrast, marks the translation as not equivalent, and synthesizes a test with Unicode inputs (e.g., `U+1F60A`, a four-byte emoji representing a single character) to confirm the inequivalence. Upon detecting the translation bug, it automatically generates a patch that uses `.chars().count()` to ensure proper handling of both ASCII and Unicode characters ( b3 ).

**Limitation 3: Repair.** Current translation repair techniques solely rely on simple feedback-driven approaches, which have proven inadequate in practical settings. Skel [77], Oxidizer [76], and Syzygy [75] utilize multi-turn iterative prompting techniques. AlphaTrans [45] adopts an execution trace-based reprompting strategy. For instance, consider scenario ( c1 ), where running the `test` method sequentially invokes methods `m1`, `m2`, and `m3`. If a bug is present in fragment `m2`, AlphaTrans reprompts all executed fragments individually without considering inter-fragment dependencies. This approach, while potentially resolving the bug in `m2`, risks introducing new functional bugs in previously correct fragments, such as `m1`. To overcome this limitation, MatchFixAgent uses code analysis and an LLM agent ( c2 ). The analyses expose dependencies such as the one between `m1` and `m2` and the agent interacts with the execution environment to execute, validate, and iteratively refine its generated patches.

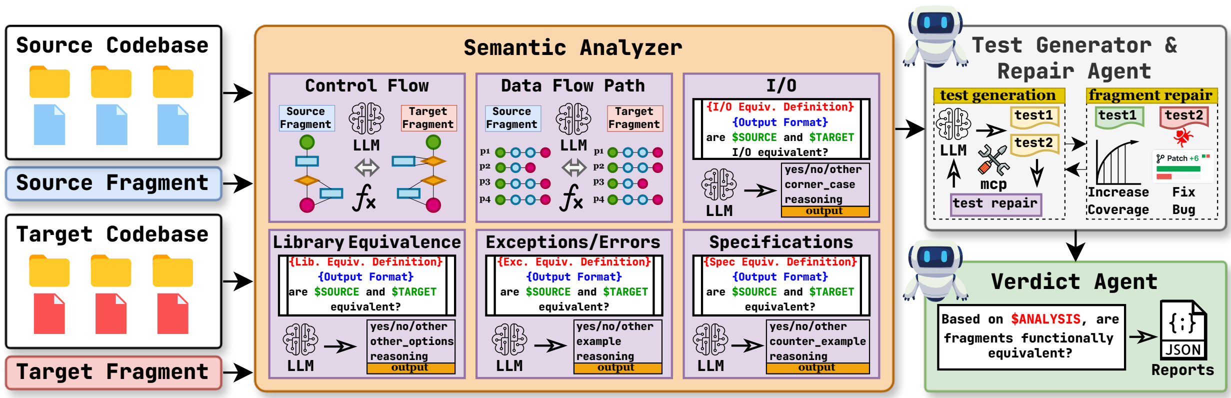



Figure 16: Overview of MatchFixAgent.

## 6.3 MatchFixAgent

### 6.3.1 Overview

Figure 16 gives an overview of MatchFixAgent, which consists of three main components: (1) the Semantic Analyzer (§6.3.2), (2) the Test Generator & Repair Agent (§6.3.3), and (3) the Verdict Agent (§6.3.4). MatchFixAgent takes as input a translation pair (source function and its translation) along with both projects, and outputs an equivalence verdict, a natural language report, and an optional repair patch. Algorithm 3 details the procedure.

The Semantic Analyzer invokes an LLM to analyze six semantic properties of the translation pair in parallel: control flow, data flow, input-output mapping, library API usage, exception handling, and specifications. This decomposition keeps the LLM focused and improves reliability [205]. Each sub-task outputs a report summarizing semantic differences. These reports are fed to the Test Generator & Repair Agent, which uses an off-the-shelf LLM coding agent (e.g., Claude Code [206] or Codex [207]) to write and execute tests that may reveal inequivalent behavior. If inequivalence is found, the agent attempts to repair the translation. Finally, the Verdict Agent validates the claims from the previous components and produces a final equivalence verdict with a summary.

**Algorithm 3** MATCHFIXAGENT

**Require:** *sourceProject*, *sourceFunc*, *translatedProject*, *translatedFunc*, *llm*, *tools*, *timeout*
**Ensure:** *validationRepairReport*

```
procedure SemanticAnalyzer(transPair, llm)
    cfgSrc, dfSrc ← build_cfg(sourceFunc)
    cfgTgt, dfTgt ← build_cfg(translatedFunc)
    return { controlFlowAnalyzer(cfgSrc, cfgTgt, transPair, llm),
             dataFlowPathAnalyzer(dfSrc, dfTgt, transPair, llm),
             ioAnalyzer(transPair, llm),
             libraryAnalyzer(transPair, llm),
             exceptionAnalyzer(transPair, llm),
             specAnalyzer(transPair, llm) }
end procedure

transPair ← [ sourceFunc, translatedFunc ]
semAnalysis ← await SemanticAnalyzer(transPair, llm)
testRepair ← testGenRepairAgent(prompt, llm, tools, timeout)
verdict ← verdictAgent(semAnalysis, testRepair, llm)
validationRepairReport ← semAnalysis ∪ testRepair ∪ verdict
return validationRepairReport
```

### 6.3.2 Semantic Analyzer

The Semantic Analyzer takes as input the translation pair and an LLM. It first computes a control flow graph (CFG) and data flow graph (DFG), and then calls six sub-analyzers in parallel, each of which analyzes a different semantic property of the translation pair. Each sub-analyzer invokes the LLM with a custom prompt [9] describing the analysis to perform. The prompts are relatively simple and short. The prompt first defines a role for the LLM (``You are an expert in...''), a general definition of functional equivalence, and a specific definition of equivalence for the semantic property. It then instructs the analyzer to output an equivalence verdict and explanation for the specific semantic property it analyzed. In addition, certain analyzers output examples to demonstrate inequivalence. The final output of the Semantic Analyzer is a 6-tuple containing a JSON-formatted output of each sub-analyzer. The following subsections provide more details on the prompts of the six sub-analyzers.

**Control Flow Analyzer.** The *Control Flow Analyzer* is prompted to analyze the

[9]Please refer to our artifacts repository [42] for prompt structure of each semantic analyzer.

```
============= Source Code ============
def max(a, b):
  """
  The function max(a, b) returns the
  greater of the two input values a and b.
  If a is greater than b, it returns a;
  otherwise, it returns a.
  """
  if a > b:
    return a
  else:
    return b
```

```
========= Control Flow Graph =========
Node 0: PARAMETERS
  parameter: a, parameter: b
  → Node 1: ENTRY
Node 1: ENTRY
  → Node 2: if a > b:
Node 2:
  if a > b:
  → Node 3 [label="a > b"]:
     return a
  → Node 4 [label="(a <= b)"]:
     return b
```

```
=========== Data Flow Paths ===========
Variable: a
  Path 1:
    CFG Node Path: 0 → 1 → 2
  Path 2:
    CFG Node Path: 0 → 1 → 2 → 3

Variable: b
  Path 1:
    CFG Node Path: 0 → 1 → 2
  Path 2:
    CFG Node Path: 0 → 1 → 2 → 4
```

Figure 17: CFG and DFP structures extracted by the Semantic Analyzer component in MatchFixAgent.

control flow structures of the source and translation to determine whether they are equivalent, looking for inequivalences such as reordered conditions, missing branches, or altered loop termination criteria. To aid this task, we provide the LLM with textual representations of the source and translation's CFGs. An example is shown in Figure 17. To compute the CFG, we use Tree-Sitter [113] to construct an abstract syntax tree of the function, and then extract basic blocks and control flow structures. Tree-sitter supports 165+ PLs, making this process PL-agnostic. Each PL supported by MatchFixAgent required approximately 280 lines of code, making it very easy to support many PLs.

To improve reliability and reduce costs, the control flow analyzer first (symbolically) computes a similarity score between the CFGs, and, if it falls above a threshold, it immediately returns an equivalent verdict without invoking the LLM. The overall procedure is shown in Algorithm 4. The analyzer abstracts each graph into canonical forms (lines 1–11) capturing node types (e.g., conditionals, loops, exception handlers) and edge types (control transfer relationships), then computes the structural similarity score based on the Jaccard index [208] (lines 12–14). We use 0.7 as the threshold. This results in approximately 25% of LLM invocations being skipped in our experiments, making this threshold relatively stringent.

**Data Flow Analyzer.** The *Data Flow Analyzer* is prompted to evaluate whether the flow of data within the source and translation is equivalent, looking for issues like unused variables. To aid the LLM, we provide textual representations of the source and translation's data flow graphs (DFGs). An example is shown in Figure 17. We keep our

**Algorithm 4** Control Flow Analyzer

```
Require: cfgSource, cfgTarget, fragments, model, τ = 0.7
Ensure: cfgAnalysis

procedure AbstractGraph(cfg)
    for (u, v) with edge ∈ cfg do
        uType, vType ← classifyNode(u), classifyNode(v)
        edgeType ← classifyEdge(edge)
        nodes ← nodes ∪ uType ∪ vType
        edges ← edges ∪ ⟨uType, edgeType, vType⟩
    end for
    return nodes, edges
end procedure

sourceNodes, sourceEdges ← AbstractGraph(cfgSource)
targetNodes, targetEdges ← AbstractGraph(cfgTarget)
nodeSim ← jaccardSimilarity(sourceNodes, targetNodes)
edgeSim ← jaccardSimilarity(sourceEdges, targetEdges)
similarity ← (0.5 × nodeSim) + (0.5 × edgeSim)
if similarity ⩾ τ then
    cfgAnalysis ← { "is_equivalent" : "yes" }
else
    cfgAnalysis ← LLM(cfgSource, cfgTarget, fragments, model)
end if
return cfgAnalysis
```

data flow computation extremely simple. For each statement in the AST, we extract variable names, label them as a def or a use, and associate them with a CFG node. We then extract def-use chains. This is primarily a syntactic analysis. We do not handle challenging problems such as aliasing, concurrency, or context sensitivity. The per-PL implementation effort is approximately 280 lines of code based on the six PLs supported by MatchFixAgent. Similar to our control flow analyzer, we compute a similarity score between the DFGs, and short-circuit if it falls above a threshold. The analyzer, show in Algorithm 5, first extracts *def-use* chains for parameters and local variables, capturing how data values are defined, propagated, and consumed. The extracted paths are compared using edit distance [209] as the similarity measure (lines 1–16). We again use 0.7 as the threshold, which results in approximately 35% of LLM invocations being skipped.

**IO Analysis.** The *IO Analyzer* is prompted to evaluate whether the observable

**Algorithm 5** Data Flow Path Analyzer

**Require:** *dfSource*, *dfTarget*, *fragments*, *model*, $\tau = 0.7$
**Ensure:** *dfAnalysis*

```
procedure COMPUTEEDITDISTANCE(srcPath, tgtPath)
    sim_a ← 0
    for all xPath ∈ srcPath do
        best ← 0
        for all yPath ∈ tgtPath do
            score ← jaccardSimilarity(xPath, yPath)
            best ← max(best, score)
        end for
        sim_a ← sim_a + best
    end for
    sim_a ← sim_a / |srcPath|
    sim_b ← repeat loop with srcPath and tgtPath swapped
    return (sim_a + sim_b) / 2
end procedure

srcPath, tgtPath ← getVariablePaths(dfSource, dfTarget)
similarity ← COMPUTEEDITDISTANCE(srcPath, tgtPath)
if similarity ⩾ τ then
    dfAnalysis ← { "is_equivalent" : "yes" }
else
    dfAnalysis ← LLM(dfSource, dfTarget, fragments, model)
end if
return dfAnalysis
```

input-output behavior of the source and target fragments is semantically equivalent. The prompt includes an IO equivalence definition, which assess five dimensions: (1) semantic equivalence of accepted inputs, (2) consistency of produced outputs, (3) preservation of side effects (e.g., file operations, network calls, or global state modifications), (4) uniform handling of edge cases, and (5) similarity in performance-critical complexity. The LLM is prompted also prompted to produce a plausible input that would trigger dissimilar IO behavior if it believes the translation is inequivalent. This methodology catches inequivalences such as differing error messages, inconsistent encoding assumptions, or missing side effects—often overlooked by structural analyses alone.

**Library Analyzer.** The *Library API Analyzer* is prompted to consider the behavior of external library APIs called in the source and translation, and evaluate whether their differences result in inequivalent behavior. This analyzer primarily detects subtle differences between similar library APIs in the source and translation. It provides suggestions to fix inequivalent behavior as well.

**Exception & Error Analyzer.** The *Exception and Error Handling Analyzer* is prompted to validate whether error detection, exception raising, and error recovery mechanisms in the source and target code fragments are functionally equivalent. The prompt include five dimensions for equivalence: (1) detecting and handling the same error conditions, (2) using semantically equivalent exception/error types, (3) producing equivalent error messages or codes, (4) preserving consistent recovery mechanisms, and (5) propagating errors in equivalent ways. If neither fragment implements explicit error handling, the analyzer deems them equivalent for this dimension. Otherwise, it statically identifies exception constructs (e.g., `try-catch`, `throw`, `return`-error patterns) and uses LLM reasoning to compare semantics. For instance, if the source raises a specific `FileNotFoundError` while the target raises a generic `IOException`, the discrepancy is flagged, as it may affect upstream handling. Where differences exist, the analyzer also recommends target-language error handling constructs that align with the source's semantics.

**Specifications Analyzer.** The *Specification Analyzer* is prompted to assess whether the source and target code fragments adhere to the same explicit or implicit functional specifications. The prompt includes Specification equivalence definition which state that the source and translation should: (1) fulfill the same documented or inferred functional requirements, (2) satisfy identical pre-conditions and post-conditions, (3) maintain the same invariants, and (4) handle the same input domain, including edge cases. The LLM is instructed to extract available specifications from function signatures, type annotations, and relevant comments, or, when no formal documentation exists, infer behavioral contracts from code semantics. The LLM is asked to compare the contracts of the source and translation. For example, if the source specifies ``` ``returns 1 on success, 0 on failure'' ``` and the target returns `Boolean` values, the inconsistency is flagged. In such cases, the LLM is instructed to produce a formalized specification that reconciles both implementations and provides counterexamples demonstrating behavior mismatch.

### 6.3.3 Test Generator & Repair Agent

The *Test Generator & Repair Agent* uses an off-the-shelf LLM coding agent and the reports from the Semantic Analyzer to write and execute tests that demonstrate functional (in)equivalence. This agent helps catch hallucinations and confirm the claims in the Semantic Analyzer reports. The prompt for the agent is shown in Figure 18, which includes a definition of equivalence, instructions to write tests in both the source and target PL that test the (in)equivalence of the translation, and finally instructions to repair the translation if it is not equivalent. We use Claude Code [206] as the agent for most of our experiments, which comes with a set of tools out of the box, namely, reading + writing files, executing arbitrary shell commands, and searching the web. The agent outputs an overall equivalence verdict, a set of tests in both the source and target PL, and a translation patch if the agent believed the translation was not equivalent.

```
<fragment_details>
  <source_fragment_details> <path to source file> <implementation of source fragment>
</source_fragment_details>
  <target_fragment_details> <path to target file> <implementation of target fragment>
</target_fragment_details>
</fragment_details>

<instruction>
  You are an expert agent specializing in test generation and code repair. Based on
  the analysis from multiple expert agents regarding functional equivalence between
  $SOURCE_LANGUAGE and $TARGET_LANGUAGE implementations of the given method/function,
  your task is to generate tests and repair the target implementation, if necessary.
  <functional_equivalence_definition>
    Two code fragments in different programming languages are considered functionally
    equivalent if, when executed on the same input, they always have identical program
    states at all corresponding points reachable by program execution, and they both
    produce the same output upon termination.
  </functional_equivalence_definition>
  <rules_and_general_notes> 1. Consider the Semantic Analyzer results ..., 2. Generate
tests ... </rules_and_general_notes>
</instruction>

<semantic_analysis_results> {"control_flow": <>, "data_flow": <>, "io": <>, "lib_api":
<>, ...} </semantic_analysis_results>

<final_response_format> {"is_equivalent": <>, "explanation": <>, "tests": <>, "patch":
<>, ...} </final_response_format>
```

Figure 18: Prompt structure of Test Generator and Repair Agent.

### 6.3.4 Verdict Agent

The final component of MatchFixAgent is the Verdict Agent, which produces a definitive assessment of the translation's correctness by synthesizing the information from the previous two stages. The Verdict Agent takes as input the semantic analysis report and the test execution + repair report. It leverages another LLM agent to consolidate these results into a final verdict. This agent's primary job is to (1) confirm the results of the Test Generator & Repair Agent, and (2) to produce a condensed summary of the results, which is useful for end-users.

## 6.4 Evaluation

We evaluate MatchFixAgent and answer the following research questions:

**RQ1:** *Effectiveness of MatchFixAgent in Translation Validation.* To what extent can MatchFixAgent automatically validate repository-level code translation? How does MatchFixAgent compare against existing validation systems?

**RQ2:** *Effectiveness of MatchFixAgent in Translation Repair.* How effective is MatchFixAgent in repairing translation bugs from real-world projects? How does the repair component compare with existing tools?

**RQ3:** *Development Cost and Adaptability.* How does the development cost and adaptability of MatchFixAgent compare to existing work? Does it work with other LLMs and agents?

**RQ4:** *Ablation Study.* How do the semantic analyzer and test generator components contribute to MatchFixAgent's effectiveness? Can a standalone code agent perform similarly to MatchFixAgent?

**RQ5:** *Additional Analysis.* How does the threshold value affect the performance of MatchFixAgent? Does an open-source model perform similar to Claude? How

Table 13: Details of benchmarks[11] from existing techniques used in MatchFixAgent. **LoC:** Lines of code in the source project.

| Tool | Project | Source Language | Target Language | Total # Trans. Pairs | LoC | Test Coverage (%) | Stars | Forks |
|---|---|---|---|---|---|---|---|---|
| Oxidizer [76] | checkdigit [210] | Go | Rust | 29 | 428 | 86.2 | 111 | 8 |
| | go-edlib [211] | | | 24 | 639 | 100 | 517 | 27 |
| | histogram [212] | | | 19 | 314 | 68.4 | 176 | 31 |
| | nameparts [213] | | | 15 | 413 | 100 | 43 | 5 |
| | stats [214] | | | 53 | 1241 | 98.1 | 2989 | 170 |
| | textrank [215] | | | 52 | 1132 | 100 | 217 | 22 |
| AlphaTrans [45] | cli [186] | Java | Python | 273 | 37841 | 93.8 | 372 | 201 |
| | csv [188] | | | 235 | 33072 | 88.1 | 392 | 278 |
| | fileupload [191] | | | 192 | 3567 | 20.8 | 246 | 185 |
| | validator [195] | | | 646 | 41605 | 62.5 | 216 | 164 |
| Skel [77] | bst [216] | Python | JavaScript | 19 | 123 | 100 | 203000 | 47000 |
| | colorsys [217] | | | 8 | 120 | 100 | 67900 | 32300 |
| | heapq [218] | | | 22 | 189 | 100 | 67900 | 32300 |
| | html [219] | | | 44 | 684 | 95.5 | 67900 | 32300 |
| | mathgen [220] | | | 81 | 735 | 100 | 711 | 183 |
| | rbt [221] | | | 27 | 366 | 96.3 | 203000 | 47000 |
| | strsim [222] | | | 64 | 654 | 78.1 | 1014 | 125 |
| | toml [223] | | | 72 | 1206 | 65.3 | 1126 | 192 |
| RustRepoTrans [88] | charset [224] | Python | Rust | 33 | 4231 | 100 | 672 | 56 |
| | deltachat [225] | C | | 125 | 23116 | 98.4 | 306 | 28 |
| | iceberg-java [226] | Java | | 25 | 592793 | 100 | 7700 | 2700 |
| | iceberg-py [227] | Python | | 44 | 49746 | 97.7 | 805 | 327 |
| | crypto-c [228] | C | | 20 | 5922 | 100 | 36 | 15 |
| | crypto-java [229] | Java | | 97 | 110261 | 100 | 2 | 7 |
| **Total** | | | | **2219** | **910398** | **89.6** | **627351** | **195624** |

much less does it cost and how to further optimize MatchFixAgent?

### 6.4.1 Experimental Setup

**Benchmark.** We evaluate MatchFixAgent on benchmarks used in prior work on automated repository-level translation. Each benchmark problem is a translation pair: the source function and the corresponding translation. The task for each benchmark problem is to give a verdict on the functional equivalence of pairs in two different PLs, and repair translation in case of equivalence. Our subjects are open-source repository-level translations with equivalence verdicts available[12] from the peer-reviewed literature[13]. We make selections across a diverse set of PL pairs.

Table 13 summarizes our subject translation pairs. We collect subjects from three

[11]Some projects in Skel (e.g., `colorsys`) are part of a bigger project [230]. The reported Star and Fork numbers belong to that bigger project.

[12]We exclude rule-based transpilers as they do not include a validation mechanism, i.e., their translation is (theoretically) correct by construction.

[13]This criterion excludes techniques such as RustMap [79] and C2SaferRust [78].

recent repository-level code translation techniques [45], [76], [77]. We did not include SYZYGY [75] because its artifact does not provide a validation system for individual functions (it only provides end-to-end tests). These works generated translations of real-world open source GitHub projects, and performed equivalence validation at the individual function level. Since ALPHATRANS [45] is evaluated on a large number of functions, we randomly sample 1346 of its total 4643 translation pairs[14].

To demonstrate the adaptability of MATCHFIXAGENT to more PL pairs, we also collect subjects from RUSTREPOTRANS [88], a benchmark consisting of human-written translations into Rust and unit tests. We exclude translation pairs collected from the libp2p [231] projects in RUSTREPOTRANS due to the presence of non-deterministic flaky tests that may result in false negatives, i.e., functional inequivalence while translation is correct, unfairly biasing comparison in favor of MATCHFIXAGENT. In total, we collect 2,219 translation pairs with over $900K$ lines of code from 24 projects and in 6 different PL pairs.

**LLMs.** Major software engineering leaderboards [232], [233] have shown that Claude Sonnet [33] outperforms other proprietary LLMs, such as OpenAI GPT-4o [234] and Google Gemini Pro [235]. Therefore, we use Anthropic's `Claude 3.7 Sonnet` [33] and Claude Code (1.0.51) [206] as the main LLM and agent in all our experiments. To show the adaptability of MATCHFIXAGENT to different LLMs and agentic frameworks, we repeat a subset of experiments using OpenAI `o4-mini-2025-04-16` [236] and Codex [207] (§6.4.4). To support future research and external validation, MATCHFIXAGENT logs the inputs, intermediate agent interactions, tool execution results, and outputs of the LLM, and supports visualizing and inspecting these logs. MATCHFIXAGENT terminates within the budget of 1,000 seconds. We empirically set this timeout after analyzing the execution time of 300 samples.

[14] ALPHATRANS translated both application and test code. In this work, we only included the application code translation pairs. While reviewing their artifacts, we also noted 11 translation pairs to be dead code and excluded them.

**Competing Validation & Repair Tools.** We compare MatchFixAgent's validation technique with the automated validation techniques proposed by Skel [77], Oxidizer [76], and AlphaTrans [45]. Except RustRepoTrans, other approaches do not explicitly report the repair results, as the repair process is interleaved with translation in the loop. For those techniques [45], [76], [77], we compare MatchFixAgent repair results with their final translation success.

**MatchFixAgent Implementation.** We implement our structure-based semantic analysis on top of Tree-Sitter [113], as it supports 165+ languages, including six PLs we target in this study. For running tests and validating patches, MatchFixAgent uses Rust 1.87.0 [237], Python 3.12.9 [238], Java 21.0.7 [239], Node 22.16.0 [240], GCC 7.3.1 [241], and Go 1.24.4 [242].

### 6.4.2 RQ1: Effectiveness of MatchFixAgent in Translation Validation

To answer this RQ, we run MatchFixAgent on each translation pair to obtain an equivalence verdict, and compare it with the verdict of existing tools. We then investigate disagreements between verdicts to determine which verdict is correct.

**Translation Validation.** To assess effectiveness in translation validation, we run MatchFixAgent on each translation pair to obtain an equivalence verdict, and compare it with the verdict of existing tools. The columns under **Tool Validation** and **MatchFixAgent** in Table 14 summarize the equivalence verdicts for the competing validation tool and MatchFixAgent, respectively. There are three types of equivalence verdicts: (1) Equivalent (**EQ**) indicating the source function and its translation are equivalent, (2) Not Equivalent (**NEQ**) indicating they are inequivalent, and (3) Validation Failure (**VF**) indicating that the tool failed to provide a verdict. Competing tools can fail to provide a verdict if (1) the source project did not have unit tests covering the function, or (2) the competing tool's language interoperability mechanism crashes before providing verdict. MatchFixAgent can fail to provide a verdict if the timeout

Table 14: Effectiveness of MatchFixAgent in translation validation compared to existing techniques. **EQ:** Equivalent, **NEQ:** Not Equivalent, **VF:** Validation Failure, **Agreement**: number and percentage of translation pairs where MatchFixAgent's and the existing tool's verdicts agree, **Disagreement**: percentage of disagreements ruled in favor of **Tool** and MatchFixAgent (**Ours**). **VF**s are excluded from **Agreement** and **Disagreement** calculations.

| Tool | Project | Total # Trans. Pairs | Tool Validation | | | MatchFixAgent | | | Agreement | Disagreement | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | EQ | NEQ | VF | EQ | NEQ | VF | | Tool | Ours |
| Oxidizer | checkdigit | 29 | 21 (72.4) | 8 (27.6) | 0 (0) | 22 (75.9) | 7 (24.1) | 0 (0) | 24 (82.8) | 0.0 | 100 |
| | go-edlib | 24 | 18 (75) | 6 (25) | 0 (0) | 16 (66.7) | 8 (33.3) | 0 (0) | 14 (58.3) | 11.1 | 88.9 |
| | histogram | 19 | 12 (63.2) | 7 (36.8) | 0 (0) | 11 (57.9) | 7 (36.8) | 1 (5.3) | 8 (44.4) | 20.0 | 80.0 |
| | nameparts | 15 | 9 (60) | 6 (40) | 0 (0) | 12 (80) | 3 (20) | 0 (0) | 12 (80) | 33.3 | 66.7 |
| | stats | 53 | 38 (71.7) | 14 (26.4) | 1 (1.9) | 37 (69.8) | 16 (30.2) | 0 (0) | 35 (67.3) | 0.0 | 100 |
| | textrank | 52 | 40 (76.9) | 12 (23.1) | 0 (0) | 34 (65.4) | 18 (34.6) | 0 (0) | 28 (53.8) | 42.9 | 57.1 |
| **Total** | | **192** | **138 (71.9)** | **53 (27.6)** | **1 (0.5)** | **132 (68.8)** | **59 (30.7)** | **1 (0.5)** | **121 (63.7)** | **15.9** | **84.1** |
| AlphaTrans | cli | 273 | 210 (76.9) | 24 (8.8) | 39 (14.3) | 210 (76.9) | 60 (22) | 3 (1.1) | 176 (76.2) | 30.0 | 70.0 |
| | csv | 235 | 97 (41.3) | 61 (26) | 77 (32.8) | 185 (78.7) | 49 (20.9) | 1 (0.4) | 108 (68.8) | 30.0 | 70.0 |
| | fileupload | 192 | 19 (9.9) | 1 (0.5) | 172 (89.6) | 144 (75) | 48 (25) | 0 (0) | 16 (80) | 25.0 | 75.0 |
| | validator | 646 | 247 (38.2) | 103 (15.9) | 296 (45.8) | 483 (74.8) | 163 (25.2) | 0 (0) | 225 (64.3) | 20.0 | 80.0 |
| **Total** | | **1346** | **573 (42.6)** | **189 (14)** | **584 (43.4)** | **1022 (75.9)** | **320 (23.8)** | **4 (0.3)** | **525 (69.3)** | **26.5** | **73.5** |
| Skel | bst | 19 | 19 (100) | 0 (0) | 0 (0) | 14 (73.7) | 5 (26.3) | 0 (0) | 14 (73.7) | 20.0 | 80.0 |
| | colorsys | 8 | 8 (100) | 0 (0) | 0 (0) | 7 (87.5) | 1 (12.5) | 0 (0) | 7 (87.5) | 0.0 | 100 |
| | heapq | 22 | 19 (86.4) | 3 (13.6) | 0 (0) | 12 (54.5) | 10 (45.5) | 0 (0) | 13 (59.1) | 50.0 | 50.0 |
| | html | 44 | 40 (90.9) | 2 (4.5) | 2 (4.5) | 35 (79.5) | 9 (20.5) | 0 (0) | 33 (78.6) | 66.7 | 33.3 |
| | mathgen | 81 | 77 (95.1) | 4 (4.9) | 0 (0) | 65 (80.2) | 16 (19.8) | 0 (0) | 67 (82.7) | 50.0 | 50.0 |
| | rbt | 27 | 26 (96.3) | 0 (0) | 1 (3.7) | 23 (85.2) | 4 (14.8) | 0 (0) | 22 (84.6) | 75.0 | 25.0 |
| | strsim | 64 | 50 (78.1) | 0 (0) | 14 (21.9) | 56 (87.5) | 8 (12.5) | 0 (0) | 44 (88) | 40.0 | 60.0 |
| | toml | 72 | 37 (51.4) | 10 (13.9) | 25 (34.7) | 49 (68.1) | 22 (30.6) | 1 (1.4) | 33 (71.7) | 40.0 | 60.0 |
| **Total** | | **337** | **276 (81.9)** | **19 (5.6)** | **42 (12.5)** | **261 (77.4)** | **75 (22.3)** | **1 (0.3)** | **233 (79.3)** | **46.5** | **53.5** |
| RustRepoTrans | charset | 33 | 20 (60.6) | 13 (39.4) | 0 (0) | 14 (42.4) | 19 (57.6) | 0 (0) | 25 (75.8) | 75.0 | 25.0 |
| | deltachat | 125 | 54 (43.2) | 69 (55.2) | 2 (1.6) | 39 (31.2) | 84 (67.2) | 2 (1.6) | 92 (76) | 90.0 | 10.0 |
| | iceberg-java | 25 | 9 (36) | 16 (64) | 0 (0) | 3 (12) | 16 (64) | 6 (24) | 15 (78.9) | 100 | 0.0 |
| | iceberg-py | 44 | 15 (34.1) | 28 (63.6) | 1 (2.3) | 11 (25) | 29 (65.9) | 4 (9.1) | 35 (89.7) | 100 | 0.0 |
| | crypto-c | 20 | 16 (80) | 4 (20) | 0 (0) | 7 (35) | 13 (65) | 0 (0) | 11 (55) | 66.7 | 33.3 |
| | crypto-java | 97 | 39 (40.2) | 58 (59.8) | 0 (0) | 30 (30.9) | 67 (69.1) | 0 (0) | 86 (88.7) | 100 | 0.0 |
| **Total** | | **344** | **153 (44.5)** | **188 (54.7)** | **3 (0.9)** | **104 (30.2)** | **228 (66.3)** | **12 (3.5)** | **264 (80.2)** | **87.5** | **12.5** |
| **Total** | | **2219** | **1140 (51.4)** | **449 (20.2)** | **630 (28.4)** | **1519 (68.5)** | **682 (30.7)** | **18 (0.8)** | **1143 (72.8)** | **39.3** | **60.7** |

limit is reached. The **Agreement** column shows the number and percentage where both MatchFixAgent's and others verdicts agree or disagree. The **Disagreement** column shows the result of our human investigation on disagreements. The **Tool** sub-column shows the percentage of disagreements where the existing tool's verdict was correct, and **Ours** sub-column shows the same metric for MatchFixAgent. On average, MatchFixAgent takes 309 seconds to produce a verdict, with an average cost of $1.22 and a total cost of $2,710.45.

These results demonstrate effectiveness of MatchFixAgent in providing an equivalence verdict: MatchFixAgent provides verdicts for 99.7% of translation pairs from AlphaTrans and Skel, whereas these techniques report verdicts for only 56.6% and 87.5% of their studied translation pairs, respectively. In addition, MatchFixAgent's verdicts show a high level of agreement with all prior work, ranging from 63.7% to 80.2%. Furthermore, we manually investigate (dis)agreements and show examples of incorrect verdicts.

**Analysis of the Disagreements.** Both MatchFixAgent and existing validation approaches are prone to false positives (i.e. the verdict is equivalent but the translation is not) and false negatives. To determine false positives and false negatives, we perform a manual investigation.

We first categorize disagreements into two cases: $D_1$, where MatchFixAgent produced a *not equivalent* verdict and the other disagreed; and $D_2$, where MatchFixAgent produced an *equivalent* verdict and the other disagreed. Due to large number of studied translation pairs, we randomly sample five instances of both $D_1$ and $D_2$ from each of the 24 projects. If a project had fewer than five disagreements, we considered all instances without sampling. Two authors[15] independently reviewed disagreements to verify whether *the not equivalent verdict* was correct (by MatchFixAgent in $D_1$ and by others in $D_2$). If the not equivalent verdict is correct, the

[15]The selected authors have total experience of 17 years in academia and 4.5 years in industry.

reviewer rules it in favor of the tool that said not equivalent, otherwise they rule it favor of the tool that said equivalent. During the investigation, 3 disagreements with OXIDIZER were due to the tool's function mocking not being enabled, and were unable to enable it. In addition, 11 disagreements with RUSTREPOTRANS were due to the correct translation not being ``1:1'', or in other words, the correct translation is not functionally equivalent to the source function. These disagreements would have been unfairly ruled in favor of MATCHFIXAGENT, and so were filtered out, leaving us with 145 disagreement cases ($D_1 = 92$, $D_2 = 53$). When the reviewer's resolution conflicted (18.6% of cases), they met with each other and agreed on the final resolution[16].

The **Disagreement** columns in Table 14 summarize the result of our human investigation. The **Tool** column shows the percentage of disagreements ruled in favor of the existing validation tool, and the **Ours** columns shows the percentage ruled in favor of MATCHFIXAGENT. The disagreement resolutions show that MATCHFIXAGENT 's verdicts are often more accurate than existing automated validation tools. MATCHFIXAGENT's verdicts are significantly more accurate than OXIDIZER and ALPHATRANS—disagreements are ruled in favor of MATCHFIXAGENT in 84.1% and 73.5% of cases, respectively. Compared to SKEL, MATCHFIXAGENT shows similar accuracy (53.5%). On RUSTREPOTRANS, MATCHFIXAGENT's accuracy fairs worse (12.5%), which is due to the relative complexity of its translation pairs.

Existing validation approaches produced 88 incorrect verdicts, 80 of which fell into three categories: (1) **Inadequate Unit Tests (42 of cases)**, (2) **Excessively Strict Equivalence Definition (11 of cases)**, and (3) **Language Interoperability Bug (27 of cases)**. A language interoperability bug means that the tool's process for converting concrete inputs in the source language to the target language did not preserve equivalence. For example, OXIDIZER's conversion of `Go runes` (which represent a Unicode character) to `Rust chars` resulted in different Unicode characters.

[16]The results of our human investigation with comments by each reviewer are publicly available [42].

On the benchmarks associated with automated validation tools (Skel, Oxidizer, AlphaTrans), MatchFixAgent produced 36 incorrect verdicts, 34 of which fell into three categories: (1) **Hallucination (23 cases)**, (2) **Inadequate Unit Tests (4 cases)** (meaning MatchFixAgent missed an input that would demonstrate inequivalence), (3) **Infeasible Input (7 cases)**. An infeasible input means that the LLM discovered an input where the source function and translation produce different outputs, but the input can never occur when the project is used as intended. Infeasible inputs often involve directly initializing private class/struct members, or calling private helper methods in unintended ways.

On RustRepoTrans's benchmarks, MatchFixAgent produced 21 incorrect verdicts. Two of the main causes are similar to the other benchmarks: (1) **Hallucination (7 cases)** and (2) **Inadequate Unit Tests (6 cases)**. The increased rates of these two causes are due to the relative complexity and size of RustRepoTrans's projects. The other major cause is **Excessively Strict Equivalence Definition (6 cases)**. As previously mentioned, RustRepoTrans's translations include refactors to make the translation more idiomatic, which creates ambiguity around the proper definition equivalence.

**Representative Examples of Incorrect Verdicts.** The following code snippet shows an example of **Inadequate Unit Tests** on a translation pair from Oxidizer. The functions both calculate the edit distance between two strings. The functions are not equivalent because `Go`'s `len()` function counts Unicode characters, whereas `Rust`'s `.len()` function counts bytes. Oxidizer incorrectly validates the `Rust` translation as equivalent because the source project's unit tests do not cover non-ASCII inputs. However, MatchFixAgent successfully generates a test with non-ASCII inputs demonstrating they are not equivalent.

```
------------------ GO SOURCE CODE ------------------
func LCSEditDistance(str1 string, str2 string)
    int {
    // ... if conditions ...
    lcs := LCS(s1, s2)
    return (len([]rune(s1)) -
            lcs) +
           (len([]rune(s2)) -
            lcs)

}
```

```
----------------- RUST TRANSLATION -----------------
pub fn lcs_edit_distance(str1: &str, str2: &str) ->
    Result<i32> {
    // ... if conditions ...
    let lcs_len = lcs(str1, str2)?;
    let edit_distance = (str1.len() as i32 -
                         lcs_len) +
                        (str2.len() as i32 -
                         lcs_len);
    Ok(edit_distance)
}
```

The next code snippet demonstrates an example of an **Infeasible Input** taken from the `textrank` project. The below `Go` function inserts a value into the map `ranks.SentenceMap` (respectively, `ranks.sentence_map` for the `Rust` translation). MatchFixAgent discovers that these functions return different values when the map is initially `{ 1 : ``SomeString'' }` (the `Go` returns 0 while the `Rust` returns 1). However, when the `textrank` is used properly via its public interface, this initial state for the map cannot occur. The map will always contain keys from 1 to $n$, where $n$ is the number of map entries. Under this precondition, the functions are equivalent.

```
------------------ GO SOURCE CODE ------------------
func addSentence(ranks *Rank, sentence
    ParsedSentence) int {
    ranks.SentenceMap[len(ranks.SentenceMap)] =
        sentence.GetOriginal()

    return len(ranks.SentenceMap) - 1

}
```

```
----------------- RUST TRANSLATION -----------------
pub(crate) fn add_sentence(ranks: &mut Rank,
    sentence: ParsedSentence) -> Result<i32, Error> {
    let sentence_id =
        ranks.sentence_map.len() as i32;
    ranks.sentence_map.insert(sentence_id,
        sentence.original.clone());
    Ok(sentence_id)
}
```

The next code snippet demonstrates an example of an **Excessively Strict Equivalence Definition** taken from the `deltachat-core` project. Both the `C` and `Rust` function retrieve a field `blobdir`. MatchFixAgent states that these are not equivalent because (1) the `Rust` function does not perform a null check, and (2) the `C` function returns a copy of `blobdir` whereas the `Rust` returns a reference. While this is true, the translation follows `Rust`'s idioms, and a developer would not care about these differences. `Rust`'s type system prevents `blobdir` from ever being null, and the `Rust` translation returns an *immutable* reference, preventing the caller from modifying the return value.

```
------------------ C SOURCE CODE -------------------
char* dc_get_blobdir(const dc_context_t* context) {
    if (context==NULL || context->magic !=
            DC_CONTEXT_MAGIC) {
        return dc_strdup(NULL);
    }
    return dc_strdup(context->blobdir);
}
```

```
----------------- RUST TRANSLATION -----------------
pub fn get_blobdir(&self) -> &Path {



    &self.inner.blobdir



}
```

**Analysis of the Agreements.** From the original experiments using `Claude 3.7 Sonnet`, we uniformly sampled at most 5 equivalent validations across 24 benchmarks and four validation systems, resulting in 110 source–target pairs. In total, 88 were correctly validated as equivalent, and 22 validated as non-equivalent by both existing tools and MatchFixAgent. The same two authors from dispute analysis performed manual investigation of equivalent cases with 95.5% inter-rater agreement and show that $\frac{106}{110}$ (96.4%) judgements were correct, meaning $\frac{4}{110}$ (3.6%) were incorrect validations, indicating a need for more rigorous testing and oversight in MatchFixAgent.

The following code snippets show an example of false positive, where both MatchFixAgent and Oxidizer validate the translation as equivalent, but the human investigator concludes otherwise. The translated Rust code incorrectly returns an error if any word is not found in `word_val_id`, while the Go source code silently handles missing words and returns zero for missing keys and continues. This difference in error handling means they behave differently when words are not in the map. Oxidizer and MatchFixAgent failed to detect this bug due to insufficient testing.

```
------------------ GO SOURCE CODE ------------------
func FindSentencesByPhrases(ranks *Rank,
    words []string) []Sentence {
    for _, i := range words {
        for _, j := range words {
            x := ranks.WordValID[i]
            y := ranks.WordValID[j]
            ...
        }
    }
    ...
    return sentences
}
```

```
----------------- RUST TRANSLATION -----------------
pub fn find_sentences_by_phrases(
    ranks: Option<&Rank>, words: &[String]) ->
        Result<Vec<Sentence>> {
    for i in words {
        for j in words {
            let x = *ranks.word_val_id.get(i)
            .ok_or_else( anyhow::
            anyhow!("Word not found: {}", i))?;
            let y = *ranks.word_val_id.get(j)
            .ok_or_else( anyhow::
            anyhow!("Word not found: {}", j))?;
            ...
        }
    }
    ...
    Ok(sentences)
}
```

The next code snippets demonstrate an example of false negative, where both MatchFixAgent and RustRepoTrans validate the translation as non-equivalent, but the human investigator concludes otherwise. Both functions compute the current time in slots since the Unix epoch by dividing the current Unix timestamp (in seconds) by `(60 * TIME_SLOT_MINUTES)`. The logic is identical, with only language-specific differences in how current time is obtained and types used (`unsign32` vs `usize`). RustRepoTrans and MatchFixAgent validate these as functionally inequivalent because there is a compilation bug somewhere else in the codebase.

```
------------------ C SOURCE CODE -------------------
    unsign32 today(void)
    {
        /* return time in slots since epoch */
        unsign32 ti = (unsign32) time(NULL);
        return ti / (60 * TIME_SLOT_MINUTES);
    }
```

```
----------------- RUST TRANSLATION -----------------
    pub fn today() -> usize {
        // Return time in slots since epoch
        let now = SystemTime::now().duration_since(
            UNIX_EPOCH).unwrap().as_secs() as usize;
        now / (60 * rom::TIME_SLOT_MINUTES)
    }
```

**Summary.** Compared with existing automated validation approaches (Skel, Oxidizer, and AlphaTrans), MatchFixAgent is more reliable at producing equivalence verdicts. MatchFixAgent provides verdicts for 99.2% of benchmarks, compared to 71.6% for existing approaches. In addition, MatchFixAgent's verdicts are more accurate than existing approaches. MatchFixAgent's verdicts agree on 72.8% of benchmarks, and on 60.7% of disagreements, human investigation shows MatchFixAgent's verdict is correct. On the more challenging benchmarks of RustRepoTrans, MatchFixAgent's verdict still show a high agreement rate, but the disagreement cases reveal for improvement.

### 6.4.3 RQ2: Effectiveness of MatchFixAgent in Translation Repair

We investigate the effectiveness of MatchFixAgent in automated repair of translation bugs, and its ability to improve code coverage of existing projects. Table 15 shows the results of this research question. To fairly evaluate the effectiveness of existing tools and MatchFixAgent in translation repair, we extract a subset of translations where both techniques generated a patch.

In total, $\frac{265}{2219}$ (11.9%) buggy translations were considered for our study. To validate patches, we used original project tests that previously failed on buggy translations, and only considered a patch correct when all failing tests passed. We did not use generated tests by MatchFixAgent to avoid bias in our evaluation. Column *Tool Repaired* shows the number of buggy translations repaired by existing techniques. Only $\frac{49}{265}$ (18.5%) bugs have been repaired by prior techniques. Except for RustRepoTrans [88], other code translation tools failed to generate correct patches to repair translation bugs. Skel [77] reprompts the same LLM for repairing bugs, but then requires a user to manually provide a fix. AlphaTrans [45] and Oxidizer [76] generates patches in the loop, however, no effectiveness was reported in their papers, and we could not repair any translation bugs using their tools. Moreover, patches by AlphaTrans and Oxidizer could not be

Table 15: Effectiveness of MatchFixAgent in translation repair compared against existing techniques. **NEQ:** Not Equivalent, **NR:** Not Reported/Repaired.

| Tool | Project | # Total Trans. Pair | Tool NEQ ∩ MatchFixAgent NEQ | Tool Repaired | MatchFixAgent Repaired | Disagreement Repaired | Coverage (Improvement) % |
|---|---|---|---|---|---|---|---|
| Oxidizer | checkdigit | 29 | 5 (17.2) | NR | 5 (100) | 2 (100) | 86.2 (0) |
| | go-edlib | 24 | 2 (8.3) | NR | 1 (50) | 4 (100) | 100 (0) |
| | histogram | 19 | 2 (10.5) | NR | 1 (50) | 2 (66.7) | 68.4 (0) |
| | nameparts | 15 | 3 (20) | NR | 1 (33.3) | 0 (0) | 100 (0) |
| | stats | 53 | 6 (11.3) | NR | 6 (100) | 5 (100) | 100 (↑**1.9**) |
| | textrank | 52 | 3 (5.8) | NR | 3 (100) | 2 (100) | 100 (0) |
| **Total** | | **192** | **21 (10.9)** | **0 (0)** | **17 (81)** | **15 (93.8)** | **92.4 (↑0.3)** |
| AlphaTrans | cli | 273 | 9 (3.3) | NR | 7 (77.8) | 3 (100) | 100 (↑**6.2**) |
| | csv | 235 | 20 (8.5) | NR | 16 (80) | 2 (100) | 100 (↑**11.9**) |
| | fileupload | 192 | 0 (0) | - | - | 2 (100) | 98.9 (↑**78.1**) |
| | validator | 646 | 34 (5.3) | NR | 23 (67.6) | 3 (100) | 99.3 (↑**36.8**) |
| **Total** | | **1346** | **63 (4.7)** | **0 (0)** | **46 (73)** | **10 (100)** | **99.6 (↑33.3)** |
| Skel | bst | 19 | 0 (0) | - | - | 4 (100) | 100 (0) |
| | colorsys | 8 | 0 (0) | - | - | 1 (100) | 100 (0) |
| | heapq | 22 | 2 (9.1) | NR | 1 (50) | 3 (100) | 100 (0) |
| | html | 44 | 1 (2.3) | NR | 0 (0) | 2 (100) | 100 (↑**4.5**) |
| | mathgen | 81 | 3 (3.7) | NR | 1 (33.3) | 2 (66.7) | 100 (0) |
| | rbt | 27 | 0 (0) | - | - | 1 (100) | 100 (↑**3.7**) |
| | strsim | 64 | 0 (0) | - | - | 3 (100) | 100 (↑**21.9**) |
| | toml | 72 | 5 (6.9) | NR | 3 (60) | 3 (100) | 100 (↑**34.7**) |
| **Total** | | **337** | **11 (3.3)** | **0 (0)** | **5 (45.5)** | **19 (95)** | **100 (↑8.1)** |
| RustRepo Trans | charset | 33 | 12 (36.4) | 5 (41.7) | 7 (58.3) | 1 (100) | 100 (0) |
| | deltachat | 125 | 60 (48) | 10 (16.7) | 11 (18.3) | 1 (100) | 100 (↑**1.6**) |
| | iceberg-java | 25 | 12 (48) | 1 (8.3) | 1 (8.3) | 0 (0) | 100 (0) |
| | iceberg-py | 44 | 25 (56.8) | 4 (16) | 6 (24) | 0 (0) | 100 (↑**2.3**) |
| | crypto-c | 20 | 4 (20) | 1 (25) | 2 (50) | 1 (100) | 100 (0) |
| | crypto-java | 97 | 57 (58.8) | 28 (49.1) | 39 (68.4) | 0 (0) | 100 (0) |
| **Total** | | **344** | **170 (49.4)** | **49 (28.8)** | **66 (38.8)** | **3 (100)** | **100 (↑0.6)** |
| **Total** | | **2219** | **265 (11.9)** | **49 (18.5)** | **134 (50.6)** | **47 (95.9)** | **98.1 (↑ 8.5)** |

validated mostly because of limitations in their validation system. For example, the following code snippets show instance `11` from the project `commons-validator` in AlphaTrans. The GraalVM-based validation system in AlphaTrans does not validate this translation as functionally equivalent, although our manual investigation indicates that the `Python` translation is correct. Therefore, the limitation in AlphaTrans is mostly due to its validation system being unable to validate LLM patches.

```
------------------ JAVA SOURCE CODE -----------------
public boolean isOn(long flag) {
    return (this.flags & flag) == flag;
}
```

```
----------------- PYTHON TRANSLATION ----------------
def isOn(self, flag: int) -> bool:

    return (self.__flags & flag) == flag
```

Column *MatchFixAgent Repaired* shows the number of translation bugs successfully repaired by our approach. In total, MatchFixAgent repaired $\frac{134}{265}$ (50.6%) of translation bugs, 32.1% more than existing reprompting-based techniques. Given the limitations in validation system of AlphaTrans discussed earlier, we manually investigate and validate patches from this tool [17]. The following code snippets

[17]Please refer to our artifacts website [42] for detailed investigation results.

demonstrate instance 276 from the project `commons-validator` in AlphaTrans in which its reprompting-based repairing could not generate a correct patch. By contrast, MatchFixAgent successfully repairs this translation bug with the help of its Library Analyzer. The report generated by this semantic analyzer indicate `"... the standard Python datetime.date class does not have a SHORT attribute ..."` which is correct. The Test Generator & Repair Agent (§6.3.3) then leverages this analysis and successfully generate a patch by replacing `SHORT` with constant `3`.

```java
------------------ JAVA SOURCE CODE -----------------
public static DateValidator DateValidator1() {
    return new DateValidator(true, DateFormat.SHORT);
}
```

```python
----------------- PYTHON TRANSLATION ----------------
def DateValidator1() -> DateValidator:
- return DateValidator(True, datetime.date.SHORT)
+ return DateValidator(True, 3)
```

Column *Disagreement Repaired* shows the number of disagreements from RQ1 (§6.4.2) which MatchFixAgent determined as *not equivalent* and successfully generated a correct patch. Of the 49 disagreements resolved in favor of MatchFixAgent, we further asked our manual investigators if the generated patch was correct or not. On average, $\frac{47}{49}$ (95.9%) of patches were validated as correct. Notice that this column cannot be directly compared with existing techniques, because they validated disagreements as functionally correct and did not generate any patches. Moreover, Column *Coverage* indicates the overall coverage for subject projects and the total improvement as a result of tests generated by MatchFixAgent. The generated tests help improve code coverage by 8.5% (from 89.6% to 98.1%). Project `commons-fileupload` from AlphaTrans sees the most improvement (78.1%), with most of its translated fragments now validated with MatchFixAgent tests.

**Summary.** MatchFixAgent patches fix 50.6% of translation bugs, 32.1% more than existing repair techniques. Its generated target PL tests addresses the inadequate test suite limitation in prior work and help improve code coverage by 8.5%.

### 6.4.4 RQ3: Development Cost and Adaptability

In this RQ, we demonstrate the development cost of MatchFixAgent and its adaptability with other LLMs and agentic frameworks.

Table 16: Development cost of MatchFixAgent compared against existing tools.

| **Tool** | **Complexity (LoC)** | **Development Time (Months)** |
|---|---|---|
| AlphaTrans | 10859 | 8 |
| Oxidizer | 19052 | 10 |
| Skel | 3843 | 6 |
| MatchFixAgent (ours) | 1650 | 0.7 |

**Development Cost.** Table 16 shows the development cost of MatchFixAgent against existing techniques. We define cost as the tool's total lines of code (LoC). As shown in the table, developing MatchFixAgent is cheap and the initial version only consists of 1,650 LoC, supporting 6 different PLs. Its dependence only on the Tree-Sitter [113] parser makes it easy to support more languages using only 280 LoC. By contrast, other tools, such as, Skel [77], AlphaTrans [45], and Oxidizer [76] only support *one* PL pair and require significant engineering effort to adapt to more languages. More precisely, MatchFixAgent is $\times 2.3$, $\times 6.6$, and $\times 11.6$ cheaper than Skel, AlphaTrans, and Oxidizer, respectively. The static nature of MatchFixAgent makes it cost-effective and scalable, achieving better performance and revealing major limitations in existing tools.

**Adaptability.** The architecture of MatchFixAgent is largely independent of any specific LLM or agent framework, making it easy to extend and integrate with more LLMs. In this research question, we investigate the extent to which replacing Anthropic's Claude 3.7 Sonnet with OpenAI `o4-mini-2025-04-16` [236], and Claude Code with Codex [207] agent, impacts the performance of MatchFixAgent. Due to the limited cost budget, we randomly sampled 96 instances from all subject projects. While sampling, we controlled for equal contribution of equivalent and non-equivalent translations, resulting in 58 and 38 samples for each, respectively.

Figure 19 illustrates the results of this study. Our analysis indicates that MatchFixAgent with Claude Code and Codex agrees 73% of the time against existing validation systems. Moreover, we also analyzed both agents' behavior in terms of

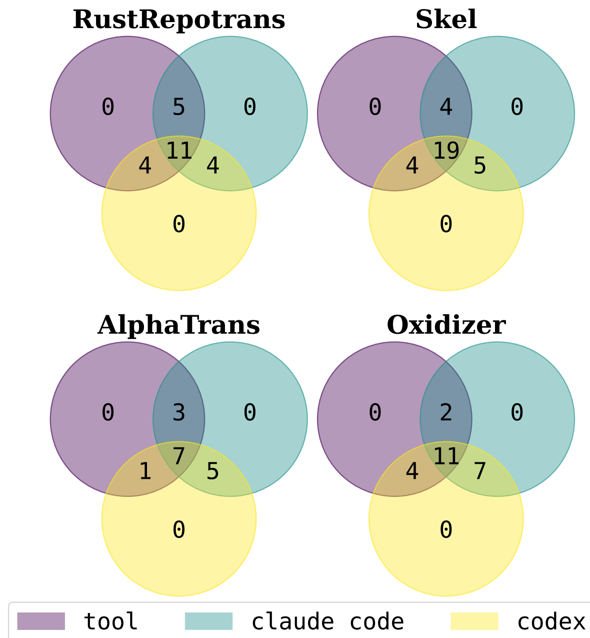



Figure 19: Agreement and dispute cases between tool validation system and MatchFix-Agent with Claude and Codex agents.

problem understanding and finding a solution. Our investigation of agent trajectories (footprint of agent actions) shows OpenAI's Codex makes fewer actions to explore the codebase, and attempts to provide a decision faster. By contrast, Anthropic's Claude Code agent first plans and reasons thoroughly about its tasks, specifically called `TODO List` by the agent, and then takes specific actions to perform each of its planned tasks.

**Summary.** MatchFixAgent is up to $\times 11.6$ times cheaper to develop than existing techniques. Also, it can be easily adapted to other LLMs and agent frameworks, e.g., OpenAI.

#### 6.4.5 RQ4: Ablation Study

We present two ablation studies to highlight the contribution of the semantic analyzer and the test generator agent in MatchFixAgent.

**Impact of Semantic Analyzer and Test Generator Agent.** To investigate the

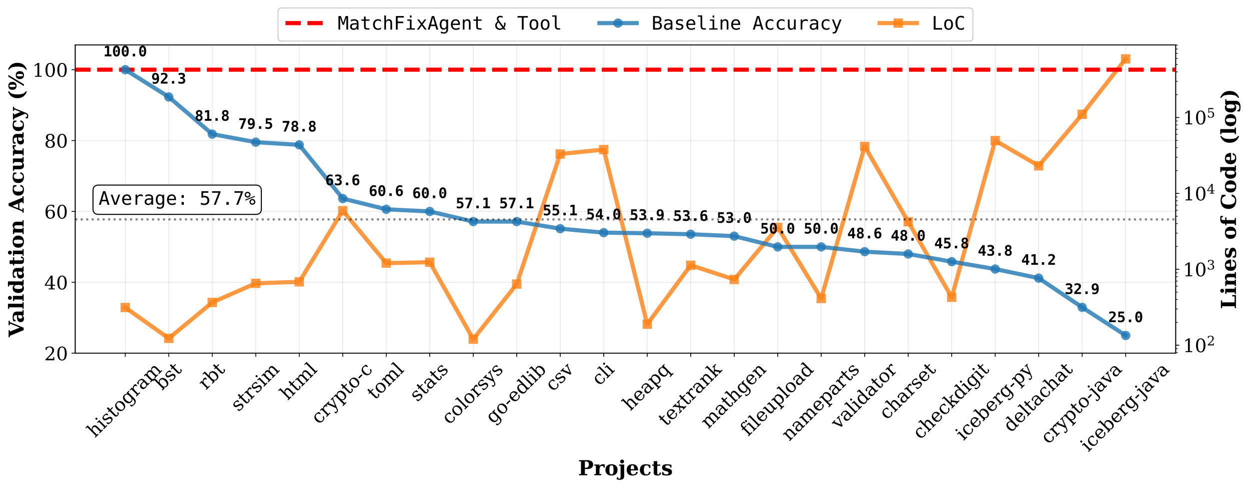



Figure 20: Impact of semantic analyzer and test generator agent in MATCHFIXAGENT. A standalone baseline agent struggles validating translations as the projects grow in size.

importance of semantic analyzer and test generator agent in MATCHFIXAGENT, we evaluated a standalone baseline agent that uses the same LLM and agent framework, e.g., `Claude Sonnet 3.7` [33] and Claude Code [206] [18]. In order to perform a controlled study, we created a sample from the original dataset where existing tools and MATCHFIXAGENT verdicts agree with each other, meaning we collected all non-dispute instances. In total, 1,091 instances, 862 equivalent, and 229 non-equivalent translations were selected. Figure 20 shows the result of this ablation study. Accuracy indicates the ratio of baseline verdicts that agree with the existing tool and MATCHFIXAGENT. On average, the validation accuracy drops by 42.3%, illustrating the importance of MATCHFIXAGENT's semantic analyzer and test generator components. Across all projects, the baseline agent only reproduced MATCHFIXAGENT and tool results in the `histogram` project from OXIDIZER [76], achieving 100% validation correctness. For the remaining projects, the agent's accuracy dropped as low as 25% in `iceberg-java`, which is the project with the largest number of lines of code.

Moreover, we also evaluated our baseline agent on dispute instances, a total of 416

[18] Please refer to our artifacts [42] for prompt templates used in this study.

cases between MatchFixAgent and existing tools. The results indicate 48.3% agreement with MatchFixAgent, and 41.1% with competing tools, however, these numbers do not convey any important meaning without ground-truths.

To address this issue, we considered the 145 cases where manual investigation established ground truth, and we can directly measure baseline agent accuracy. On the 145 manually resolved cases, the baseline agent achieves only 47.7% accuracy on the 88 cases where MatchFixAgent is correct–the cases that represent MatchFixAgent's unique advantage over existing tools. By contrast, the baseline achieves 56.1% on the 57 cases where the existing tool is correct, confirming that the baseline agent systematically fails on precisely the hard, ambiguous translations that MatchFixAgent 's semantic analyzer and test generation are designed to handle.

The 47.7% accuracy on difficult, disputed cases (where our original ablation showed only a 42.3% drop on easier, non-disputed cases) confirms that MatchFixAgent's multi-agent architecture is essential, not just helpful. The baseline agent's near-random performance on these difficult cases demonstrates that the semantic analyzer and test generation components are the core drivers of MatchFixAgent's accuracy advantage in resolving both straightforward and challenging translations.

**Impact of Semantic Analyzer.** We perform another study by removing only the semantic analysis results when prompting the test generator and verdict agents. We use the same set with 1,091 instances from the previous ablation to perform this study. Figure 21 illustrates the result of our second ablation. The results indicate that the performance of MatchFixAgent significantly drops by 39.7% without the six semantic analyses. Furthermore, we observe that the test generator agent without semantic analyzer is more costly and on average spends 3.9% (66.3$K$ instead of 63.7$K$), 6.2% (136$K$ instead of 128$K$), 4.2% (9.4$M$ instead of 9.0$M$), and 7.5% (61.28 h instead of 56.71 h) more turns/interactions, input tokens, output tokens, and time, respectively.

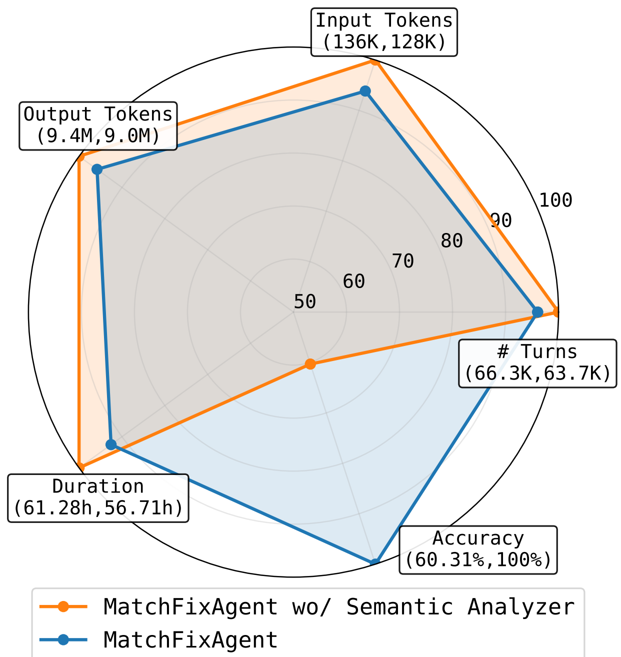



Figure 21: Removing the semantic analyzer decreases the effectiveness of MatchFixAgent, while increasing token consumption, number of turns, and processing time.

**Summary.** Removing the semantic analyzer and test generator significantly drops the effectiveness of MatchFixAgent by 42.3%. Moreover, removing the semantic analyzer alone decreases the accuracy by 39.7%, and increases the test generator cost, on average, by 5.4%.

### 6.4.6 RQ5: Additional Analysis

**Threshold Ablation.** Table 17 shows the results of our ablation study on the threshold $\tau$ in Algorithm 4 and Algorithm 5. The threshold $\tau$ governs a tradeoff between algorithmic reliance and LLM invocation cost in the Control Flow and Data Flow Path analyzers, which are the only two sub-analyzers with a short-circuit mechanism; the remaining four (I/O Mapping, Library API, Exception Handling, Specifications) are unaffected and serve as an internal control. A lower $\tau$ is easier to satisfy, causing the short-circuit to fire more frequently and reducing LLM cost, but over-trusting the structural similarity algorithm: at $\tau$=0.5, TNR on Control Flow and Data Flow Path drops to 19.7% and 17.7% respectively—meaning the structural algorithm alone fails to correctly identify inequivalent pairs in roughly four out of five cases. A higher $\tau$ is harder to satisfy, so fewer cases are short-circuited and more LLM invocations are triggered, increasing cost. Moving from $\tau$=0.5 to $\tau$=0.7 yields the largest correctness gains: TNR on Control Flow jumps +14.6% at a TPR cost of only −2.6%, and TNR on Data Flow Path gains +5.6% at a negligible −0.2% TPR cost. Moving further to $\tau$=0.9 yields only marginal additional TNR improvements (+8.4% and +3.5% respectively) while TPR continues to erode (−1.6% and −1.4%) and LLM invocation cost increases. The average TNR across all six analyzers follows the same pattern: 34.3% at $\tau$=0.5, 37.8% at $\tau$=0.7, and 39.8% at $\tau$=0.9, confirming that $\tau$=0.7 captures the steepest part of the improvement curve. We therefore select $\tau$=0.7 as the main threshold value—delivering substantial TNR improvement over $\tau$=0.5 without the diminishing returns and added cost of $\tau$=0.9. Nonetheless, since the threshold functions as a cost-optimization lever rather than a correctness lever—with any errors in a single sub-analyzer recoverable by the downstream Test Generator & Repair Agent and Verdict Agent—practitioners can adjust $\tau$ based on their own risk tolerance and computational budget, using the TPR/TNR/FPR/FNR results in Table 17.

**MatchFixAgent with Open Source Model.** MatchFixAgent is LLM-agnostic and easily adapts to new models. To show its open-source adaptability, we reproduced Table 14 with the state-of-the-art `Qwen3-Next-80B-A3B` (released February 2026). Table 18 shows the results of this experiment, indicating that the agreement rate of Qwen3 (71.6%) remains consistent with Claude (72.8%), suggesting similarity between the open-source and closed LLMs.

**Cost Analysis.** Table 19 shows the cost analysis of different components in MatchFixAgent for two models, namely, `Claude 3.7 Sonnet` and `Qwen3-Next-80B-A3B`. The results indicate the open-source Qwen3 model is on average $\times$23.8 cheaper than

Table 17: Agreement between each semantic analyzer and MatchFixAgent verdict. Experiments repeated with the Claude model for threshold $\tau \in \{0.5, 0.7, 0.9\}$. **TP:** True Positive (Semantic Analyzer=Yes and MatchFixAgent =Yes), **FP:** False Positive (Semantic Analyzer=Yes and MatchFixAgent =No), **FN:** False Negative (Semantic Analyzer=No and MatchFixAgent =Yes), **TN:** True Negative (Semantic Analyzer=No and MatchFixAgent =No).

| **Semantic Analyzer** | $\tau$ | **TP** | | **FP** | | **FN** | | **TN** | |
|---|---|---|---|---|---|---|---|---|---|
| | | **Count** | **TPR** | **Count** | **FPR** | **Count** | **FNR** | **Count** | **TNR** |
| Control Flow | 0.5 | 1396 | 0.957 | 549 | 0.803 | 63 | 0.043 | 135 | 0.197 |
| | 0.7 | 1404 | 0.931 | 438 | 0.657 | 104 | 0.069 | 229 | 0.343 |
| | 0.9 | 1342 | 0.915 | 375 | 0.573 | 124 | 0.085 | 279 | 0.427 |
| Data Flow Path | 0.5 | 1411 | 0.968 | 552 | 0.823 | 47 | 0.032 | 119 | 0.177 |
| | 0.7 | 1459 | 0.966 | 510 | 0.767 | 52 | 0.034 | 155 | 0.233 |
| | 0.9 | 1395 | 0.952 | 483 | 0.732 | 70 | 0.048 | 177 | 0.268 |
| Input/Output Mapping | 0.5 | 1399 | 0.961 | 323 | 0.474 | 57 | 0.039 | 358 | 0.526 |
| | 0.7 | 1437 | 0.953 | 321 | 0.481 | 71 | 0.047 | 347 | 0.519 |
| | 0.9 | 1416 | 0.963 | 322 | 0.481 | 55 | 0.037 | 348 | 0.519 |
| Library API Equivalence | 0.5 | 1441 | 0.992 | 478 | 0.714 | 12 | 0.008 | 191 | 0.286 |
| | 0.7 | 1481 | 0.988 | 482 | 0.730 | 18 | 0.012 | 178 | 0.270 |
| | 0.9 | 1450 | 0.989 | 477 | 0.726 | 16 | 0.011 | 180 | 0.274 |
| Exception/Error Handling | 0.5 | 1396 | 0.960 | 433 | 0.640 | 58 | 0.040 | 244 | 0.360 |
| | 0.7 | 1420 | 0.940 | 425 | 0.636 | 91 | 0.060 | 243 | 0.364 |
| | 0.9 | 1407 | 0.959 | 428 | 0.641 | 60 | 0.041 | 240 | 0.359 |
| Specifications | 0.5 | 1378 | 0.953 | 329 | 0.486 | 68 | 0.047 | 348 | 0.514 |
| | 0.7 | 1406 | 0.938 | 308 | 0.462 | 93 | 0.062 | 359 | 0.538 |
| | 0.9 | 1376 | 0.947 | 308 | 0.460 | 77 | 0.053 | 362 | 0.540 |
| **Average** | 0.5 | 1403 | 0.965 | 444 | 0.657 | 50 | 0.035 | 232 | 0.343 |
| | 0.7 | 1434 | 0.953 | 414 | 0.622 | 71 | 0.047 | 251 | 0.378 |
| | 0.9 | 1397 | 0.954 | 398 | 0.602 | 67 | 0.046 | 264 | 0.398 |

Table 18: Effectiveness of MatchFixAgent in translation validation compared to existing techniques using `Qwen3-Next-80B-A3B`. **EQ:** Equivalent, **NEQ:** Not Equivalent, **VF:** Validation Failure, **Agreement**: number and percentage of translation pairs where MatchFixAgent's and the existing tool's verdicts agree, **Disagreement**: percentage of disagreements ruled in favor of **Tool** and MatchFixAgent (**Ours**). **VF**s are excluded from **Agreement** and **Disagreement** calculations.

| Tool | Project | Total # Trans. Pairs | Tool Validation | | | MatchFixAgent | | | Agreement |
|---|---|---|---|---|---|---|---|---|---|
| | | | EQ | NEQ | VF | EQ | NEQ | VF | |
| Oxidizer | checkdigit | 29 | 21 (72.4) | 8 (27.6) | 0 (0) | 23 (79.3) | 6 (20.7) | 0 (0) | 19 (65.5) |
| | go-edlib | 24 | 18 (75) | 6 (25) | 0 (0) | 14 (58.3) | 10 (41.7) | 0 (0) | 12 (50) |
| | histogram | 19 | 12 (63.2) | 7 (36.8) | 0 (0) | 14 (73.7) | 5 (26.3) | 0 (0) | 7 (36.8) |
| | nameparts | 15 | 9 (60) | 6 (40) | 0 (0) | 10 (66.7) | 5 (33.3) | 0 (0) | 14 (93.3) |
| | stats | 53 | 38 (71.7) | 14 (26.4) | 1 (1.9) | 35 (66) | 17 (32.1) | 1 (1.9) | 33 (64.7) |
| | textrank | 52 | 40 (76.9) | 12 (23.1) | 0 (0) | 32 (61.5) | 20 (38.5) | 0 (0) | 32 (61.5) |
| **Total** | | **192** | **138 (71.9)** | **53 (27.6)** | **1 (0.5)** | **128 (66.7)** | **63 (32.8)** | **1 (0.5)** | **117 (61.6)** |
| AlphaTrans | cli | 273 | 210 (76.9) | 24 (8.8) | 39 (14.3) | 214 (78.4) | 53 (19.4) | 6 (2.2) | 173 (75.2) |
| | csv | 235 | 97 (41.3) | 61 (26) | 77 (32.8) | 155 (66) | 75 (31.9) | 5 (2.1) | 93 (60) |
| | fileupload | 192 | 19 (9.9) | 1 (0.5) | 172 (89.6) | 143 (74.5) | 43 (22.4) | 6 (3.1) | 17 (85) |
| | validator | 646 | 247 (38.2) | 103 (15.9) | 296 (45.8) | 475 (73.5) | 158 (24.5) | 13 (2) | 223 (65) |
| **Total** | | **1346** | **573 (42.6)** | **189 (14)** | **584 (43.4)** | **987 (73.3)** | **329 (24.4)** | **30 (2.2)** | **506 (67.6)** |
| Skel | bst | 19 | 19 (100) | 0 (0) | 0 (0) | 13 (68.4) | 5 (26.3) | 1 (5.3) | 13 (72.2) |
| | colorsys | 8 | 8 (100) | 0 (0) | 0 (0) | 8 (100) | 0 (0) | 0 (0) | 8 (100) |
| | heapq | 22 | 19 (86.4) | 3 (13.6) | 0 (0) | 19 (86.4) | 3 (13.6) | 0 (0) | 20 (90.9) |
| | html | 44 | 40 (90.9) | 2 (4.5) | 2 (4.5) | 23 (52.3) | 19 (43.2) | 2 (4.5) | 24 (60) |
| | mathgen | 81 | 77 (95.1) | 4 (4.9) | 0 (0) | 66 (81.5) | 12 (14.8) | 3 (3.7) | 66 (84.6) |
| | rbt | 27 | 26 (96.3) | 0 (0) | 1 (3.7) | 19 (70.4) | 5 (18.5) | 3 (11.1) | 18 (78.3) |
| | strsim | 64 | 50 (78.1) | 0 (0) | 14 (21.9) | 56 (87.5) | 6 (9.4) | 2 (3.1) | 45 (91.8) |
| | toml | 72 | 37 (51.4) | 10 (13.9) | 25 (34.7) | 46 (63.9) | 23 (31.9) | 3 (4.2) | 35 (77.8) |
| **Total** | | **337** | **276 (81.9)** | **19 (5.6)** | **42 (12.5)** | **250 (74.2)** | **73 (21.7)** | **14 (4.2)** | **229 (80.9)** |
| RustRepoTrans | charset | 33 | 20 (60.6) | 13 (39.4) | 0 (0) | 11 (33.3) | 21 (63.6) | 1 (3) | 20 (62.5) |
| | deltachat | 125 | 54 (43.2) | 69 (55.2) | 2 (1.6) | 30 (24) | 91 (72.8) | 4 (3.2) | 87 (73.1) |
| | iceberg-java | 25 | 9 (36) | 16 (64) | 0 (0) | 4 (16) | 20 (80) | 1 (4) | 20 (83.3) |
| | iceberg-py | 44 | 15 (34.1) | 28 (63.6) | 1 (2.3) | 6 (13.6) | 36 (81.8) | 2 (4.5) | 34 (82.9) |
| | crypto-c | 20 | 16 (80) | 4 (20) | 0 (0) | 10 (50) | 7 (35) | 3 (15) | 14 (82.4) |
| | crypto-java | 97 | 39 (40.2) | 58 (59.8) | 0 (0) | 26 (26.8) | 68 (70.1) | 3 (3.1) | 82 (87.2) |
| **Total** | | **344** | **153 (44.5)** | **188 (54.7)** | **3 (0.9)** | **87 (25.3)** | **243 (70.6)** | **14 (4.1)** | **257 (78.6)** |
| **Total** | | **2219** | **1140 (51.4)** | **449 (20.2)** | **630 (28.4)** | **1452 (65.4)** | **708 (31.9)** | **59 (2.7)** | **1109 (71.6)** |

Table 19: Cost analysis of MatchFixAgent components. Tuple entries indicate ⟨`Claude 3.7 Sonnet`, `Qwen3-Next-80B-A3B`⟩.

| Tool | Total # Trans. Pairs | Component | Avg Input Tokens | Avg Output Tokens | Avg Duration (ms) | Avg Cost ($) |
|---|---|---|---|---|---|---|
| AlphaTrans | 1346 | Control Flow | ⟨437.0, 349.5⟩ | ⟨86.9, 65.4⟩ | ⟨2557.5, 2557.5⟩ | ⟨0.00261, 0.00008⟩ |
| | | Data Flow | ⟨1124.5, 931.8⟩ | ⟨122.8, 93.9⟩ | ⟨3507.8, 3507.8⟩ | ⟨0.00522, 0.00016⟩ |
| | | IO | ⟨782.4, 646.0⟩ | ⟨205.0, 217.5⟩ | ⟨7059.7, 7059.7⟩ | ⟨0.00542, 0.00023⟩ |
| | | Lib Equivalence | ⟨700.4, 583.0⟩ | ⟨235.9, 218.2⟩ | ⟨6900.2, 6900.2⟩ | ⟨0.00564, 0.00022⟩ |
| | | Exception Error | ⟨726.4, 599.0⟩ | ⟨152.0, 182.2⟩ | ⟨6250.3, 6250.3⟩ | ⟨0.00446, 0.00020⟩ |
| | | Spec | ⟨726.4, 599.0⟩ | ⟨311.1, 318.6⟩ | ⟨8801.8, 8801.8⟩ | ⟨0.00685, 0.00030⟩ |
| | | Test Gen & Repair | ⟨161.6, 2104327.0⟩ | ⟨11527.8, 7672.8⟩ | ⟨274732.6, 294980.7⟩ | ⟨0.17340, 0.19537⟩ |
| | | Verdict | ⟨36.6, 32582.1⟩ | ⟨1404.6, 240.2⟩ | ⟨31737.7, 9770.2⟩ | ⟨0.02118, 0.00312⟩ |
| Oxidizer | 192 | Control Flow | ⟨1426.4, 1186.9⟩ | ⟨230.8, 168.5⟩ | ⟨3957.4, 3957.4⟩ | ⟨0.00774, 0.00024⟩ |
| | | Data Flow | ⟨2298.7, 1928.8⟩ | ⟨206.6, 159.1⟩ | ⟨3726.5, 3726.5⟩ | ⟨0.00999, 0.00030⟩ |
| | | IO | ⟨993.8, 833.6⟩ | ⟨238.7, 297.3⟩ | ⟨6532.9, 6532.9⟩ | ⟨0.00656, 0.00031⟩ |
| | | Lib Equivalence | ⟨912.8, 770.6⟩ | ⟨265.7, 250.3⟩ | ⟨5640.9, 5640.9⟩ | ⟨0.00672, 0.00026⟩ |
| | | Exception Error | ⟨937.8, 786.6⟩ | ⟨248.3, 240.8⟩ | ⟨6100.1, 6100.1⟩ | ⟨0.00654, 0.00026⟩ |
| | | Spec | ⟨937.8, 786.6⟩ | ⟨399.7, 408.7⟩ | ⟨10097.8, 10097.8⟩ | ⟨0.00881, 0.00039⟩ |
| | | Test Gen & Repair | ⟨165.2, 2415982.5⟩ | ⟨11340.1, 8769.0⟩ | ⟨244057.2, 281051.7⟩ | ⟨0.17060, 0.22428⟩ |
| | | Verdict | ⟨43.1, 28828.1⟩ | ⟨1813.4, 264.8⟩ | ⟨37238.3, 6736.6⟩ | ⟨0.02733, 0.00280⟩ |
| RustRepoTrans | 344 | Control Flow | ⟨2247.7, 1757.5⟩ | ⟨266.4, 187.0⟩ | ⟨5437.8, 5437.8⟩ | ⟨0.01074, 0.00030⟩ |
| | | Data Flow | ⟨5083.8, 4841.3⟩ | ⟨275.7, 186.1⟩ | ⟨4229.3, 4229.3⟩ | ⟨0.01939, 0.00058⟩ |
| | | IO | ⟨1167.0, 939.1⟩ | ⟨279.0, 307.9⟩ | ⟨7104.9, 7104.9⟩ | ⟨0.00769, 0.00032⟩ |
| | | Lib Equivalence | ⟨1086.0, 876.1⟩ | ⟨318.4, 275.4⟩ | ⟨6532.2, 6532.2⟩ | ⟨0.00803, 0.00029⟩ |
| | | Exception Error | ⟨1111.0, 892.1⟩ | ⟨287.8, 253.5⟩ | ⟨6326.0, 6326.0⟩ | ⟨0.00765, 0.00028⟩ |
| | | Spec | ⟨1111.0, 889.8⟩ | ⟨491.5, 457.2⟩ | ⟨9488.8, 9488.8⟩ | ⟨0.01071, 0.00044⟩ |
| | | Test Gen & Repair | ⟨189.7, 1329259.2⟩ | ⟨15020.1, 4355.7⟩ | ⟨401964.1, 148608.3⟩ | ⟨0.22587, 0.12303⟩ |
| | | Verdict | ⟨44.3, 84397.6⟩ | ⟨1925.5, 327.8⟩ | ⟨44992.7, 10575.6⟩ | ⟨0.02902, 0.00785⟩ |
| Skel | 337 | Control Flow | ⟨1163.1, 949.7⟩ | ⟨201.4, 153.5⟩ | ⟨3640.0, 3640.0⟩ | ⟨0.00651, 0.00021⟩ |
| | | Data Flow | ⟨3621.2, 4951.7⟩ | ⟨258.6, 195.3⟩ | ⟨4867.5, 4867.5⟩ | ⟨0.01474, 0.00060⟩ |
| | | IO | ⟨907.7, 761.1⟩ | ⟨218.5, 245.6⟩ | ⟨5834.6, 5834.6⟩ | ⟨0.00600, 0.00026⟩ |
| | | Lib Equivalence | ⟨826.7, 698.1⟩ | ⟨233.5, 221.2⟩ | ⟨5237.1, 5237.1⟩ | ⟨0.00598, 0.00024⟩ |
| | | Exception Error | ⟨851.7, 714.1⟩ | ⟨157.5, 192.7⟩ | ⟨4896.8, 4896.8⟩ | ⟨0.00492, 0.00021⟩ |
| | | Spec | ⟨851.7, 714.1⟩ | ⟨368.4, 378.8⟩ | ⟨8539.1, 8539.1⟩ | ⟨0.00808, 0.00036⟩ |
| | | Test Gen & Repair | ⟨160.8, 1843194.0⟩ | ⟨11803.7, 8944.5⟩ | ⟨245430.8, 272173.9⟩ | ⟨0.17754, 0.17286⟩ |
| | | Verdict | ⟨38.8, 57464.5⟩ | ⟨1421.1, 355.8⟩ | ⟨37516.6, 9687.9⟩ | ⟨0.02143, 0.00545⟩ |
| Average | 2219 | Control Flow | ⟨913.6, 731.4⟩ | ⟨144.6, 106.6⟩ | ⟨3289.5, 3289.5⟩ | ⟨0.00491, 0.00015⟩ |
| | | Data Flow | ⟨2219.1, 2234.6⟩ | ⟨174.4, 129.2⟩ | ⟨3845.1, 3845.1⟩ | ⟨0.00927, 0.00030⟩ |
| | | IO | ⟨879.3, 725.1⟩ | ⟨221.4, 242.7⟩ | ⟨6835.0, 6835.0⟩ | ⟨0.00596, 0.00025⟩ |
| | | Lib Equivalence | ⟨797.7, 662.1⟩ | ⟨250.9, 230.3⟩ | ⟨6481.6, 6481.6⟩ | ⟨0.00616, 0.00024⟩ |
| | | Exception Error | ⟨823.3, 678.1⟩ | ⟨182.2, 199.9⟩ | ⟨6043.5, 6043.5⟩ | ⟨0.00520, 0.00022⟩ |
| | | Spec | ⟨823.3, 677.8⟩ | ⟨355.4, 357.0⟩ | ⟨8980.5, 8980.5⟩ | ⟨0.00780, 0.00034⟩ |
| | | Test Gen & Repair | ⟨166.1, 1971480.1⟩ | ⟨12094.9, 7446.6⟩ | ⟨287352.3, 267620.4⟩ | ⟨0.18192, 0.18324⟩ |
| | | Verdict | ⟨38.7, 44068.9⟩ | ⟨1523.3, 273.4⟩ | ⟨35146.1, 9620.0⟩ | ⟨0.02296, 0.00418⟩ |

Claude. Please note that we rely on the numbers reported by Claude Code for input and output tokens, and calculate dollar ($) cost accordingly. For instance Claude requires $3 / 1 million input tokens and $15 / 1 million output tokens, and Qwen3 requires $0.09 / 1 million input tokens and $0.78 / 1 million output tokens. Since we run our Qwen3 experiments using a proxy server (e.g., LiteLLM [243]) for compatibility purposes, the number of cached input tokens were not reported properly and as a result there is a significant incorrect difference, for instance, $2M$ compared to 166 average input tokens for Test Generator and Repair agent across all tools. However, after checking the logs from the LLM provider, we noticed that the cost was correctly reflected, in the same order of magnitude, for the open-source Qwen3 model.

**Optimizing MatchFixAgent.** There are many directions that can be explored to optimize both the dollar cost and the runtime of MatchFixAgent, which we outline below, but leave this as future work. However, please note though that MatchFixAgent 's current dollar cost and runtime compare favorably to the development cost of existing repository-level validation techniques, or even when compared to a salaried human software engineer manually performing validation. Compared to these two baselines, 309s and $1.22 per translation pair are not daunting. A fully-automated translation tool that runs for a week and costs a few $100 is still extremely useful in practice if the translation produced is high quality. The following are potential optimizations:

1. Multiple instances of MatchFixAgent can run concurrently on functions that are not dependent on each other. We can determine function dependencies by constructing a call graph, which is a common program analysis technique. This approach would reduce the total runtime and does not require any changes to the design of MatchFixAgent.

2. Use faster, cheaper LLMs or skip analysis for ”trivial” translation pairs (e.g., simple `getter` / `setter` functions or those using only primitive types). Heuristics could detect these to avoid expensive semantic analysis or use a cheaper LLM.

3. A single instance of MatchFixAgent can process multiple related functions at once. Part of the overhead of running a coding agent is when the agent “orients” itself by reading code files to understand the codebase. By processing multiple related functions at once, we eliminate redundant ”orientation” phases, reducing both runtime and cost.

### 6.5 Related Work

**Translation Validation and Repair.** Existing automated translation validation techniques either rely on test execution [22], [29], [45], [47], [50], [75], [76], [86], [87], [88], [89], [109], [244], formal methods [84], [85], [245], or fuzzing [94] for translation validation. Abid et al. [92] and AlphaTrans [45] leverage GraalVM [91] and language interoperability to execute code in both source and target PL for translation validation. Other tools like Oxidizer [76] and Syzygy [75] instrument programs to extract input-output pairs and use in target PL for validation. Skel [77] validates translations through test execution by converting Python tests to JavaScript simply through string replacement. This suffices since source Python tests are simple function calls and value comparisons. For automated translation repair, most tools [22], [45], [75], [76], [77], [86], [88] rely on simple reprompting of LLMs with execution feedback, which has proven ineffective. Specifically, AlphaTrans [45] performs independent reprompting of suspicious fragments based on execution trace, while Skel [77] requires a user for manually repairing translation bugs.

**LLM Agents.** With the increasing prominence of agent-based frameworks [246], [247], recent research and industrial efforts have turned towards leveraging these frameworks to address various software engineering tasks [97], [98], [99]. SWE-agent [100] introduces a specialized agent-computer interface (ACI), facilitating agent interaction with code repositories via file reading, editing, and execution of bash commands. AutoCodeRover [101], which provides LLM agents with specialized code-search APIs, enables iterative retrieval and localization of code segments associated with bugs. SpecRover [102] enhances AutoCodeRover by emphasizing specification inference, generating function summaries, and providing targeted feedback at crucial agent execution stages. Agentless [103] further shows simple LLM agents can fix real-world bugs without using excessive tools and model complex environment behavior. Besides these state-of-the-art frameworks, numerous additional agent-based approaches exist both in open-source [105], [106], [135], [248], [249] and commercial products [104], [107], [108].

## 6.6 Threats to Validity

Similar to prior techniques, MatchFixAgent comes with some limitations and threats to the validity. In this section, we discuss how we mitigated various threats.

**Internal Validity.** There are two main threats to internal validity. First, we only run experiments once. Since LLMs are inherently non-deterministic, running experiments again may produce different results. While it is highly *likely* some individual equivalence verdicts and repair results would change if experiments were run again, it is highly *unlikely* the aggregate metrics we report would change significantly given the large number of translation samples we use (2,219 pairs). Second, our human investigation does not assess ground truth equivalence. We only assess whether an inequivalent verdict was correct, but we do not analyze the correctness of equivalent verdicts. While this means we don't have any measure of true accuracy of MatchFixAgent, we still can claim MatchFixAgent is more accurate than existing automated validation techniques.

**External Validity.** One main external threat is the generalizability of our approach. Our validation and repair system is very generic and can be extended to more PL pairs with minimal engineering effort. Also, the majority of tools that we used, for example, Tree-Sitter [113] can support a large set of PLs. To mitigate external validity, we built the initial version of MatchFixAgent with six PLs.

**Construct Validity.** In order to minimize construct validity, MatchFixAgent is built on well-known and rigorously tested tools, e.g., Tree-Sitter [113], Claude Code [206], and Codex [207].

## 6.7 Conclusion

We presented MatchFixAgent, a *language-agnostic* technique that combines the power of program analysis and LLM agents for autonomous repository-level code translation validation and repair. It performs various semantic analyses to systematically generate targeted tests, enabling demonstration of functional equivalence or detection of semantic bugs. Through rigorous evaluation on multiple benchmarks and different PL-pairs, we show the effectiveness of MatchFixAgent. Our manual investigation of generated reports reveals significant limitation of existing techniques. To our knowledge, MatchFixAgent is the first approach that can effectively validate and repair translations in repository-level across multiple PLs.

# Chapter 7 RECODEAGENT: A Multi-Agent Workflow for Language-agnostic Translation and Validation of Large-scale Repositories

## 7.1 Introduction

Repository-level code translation—the process of converting an entire codebase from one programming language (PL) to another—is critical to improving software reliability and security and minimizing technical debt [1], [2], [3], [5]. Early work developed rule-based approaches [18], [19], [20], [21], like C2Rust, where all translation rules are written by hand. Later work developed neuro-symbolic techniques [45], [75], [76], [77], [78], [79], [80], [81], [82], [83], [87], [90], which combine large language models (LLMs) with (symbolic) program analysis and testing. More recently, agentic approaches have been evaluated [109], [110], [111], [112], wherein one or more LLM agents work together to translate code between specific source-target PLs. Yet existing agentic systems have yet to match the translation quality of heavily engineered neuro-symbolic pipelines. More importantly, prior work offers no evidence that monolithic agent scaffolds—a single LLM agent with generic exploration tools [109], or a small number of agents without phase-specific division of labor [110], [111]—can recover comparable effectiveness simply by accumulating more tools or longer prompts. Repository-level translation instead decomposes into qualitatively distinct phases—analysis, planning, translation, and validation—each requiring its own agent scaffold with dedicated objectives and tooling. Without such principled division of labor, neither prior agentic techniques nor ad hoc single-agent instantiations of the same scaffold can approach the effectiveness of RECODEAGENT's multi-agent architecture (§7.4.4).

Prior rule-based and neuro-symbolic translation techniques only evaluate on a single source-target PL pair [18], [45], [75], [76], [77], [78], [86], [87], [89]. This is due to the enormous engineering effort required to support a PL as a source or target language. The implementation of rule-based tools can go over $100K$ LoC[19] and neuro-symbolic tools over 10K LoC[20] just to support a single source-target PL pair. Given the quadratic number of PL pairs, scaling these techniques to many PL pairs is impractical. A PL-agnostic approach can help translate and validate projects across multiple PL-pairs without the need for complex engineering and external third-party dependencies.

Theoretically, agents can enable PL-agnostic code translation. However, having an end-to-end agentic code translation and validation pipeline can be challenging due to the following limitations:

**(1) True PL-agnosticism.** Existing agentic scaffolds operate on iterative reasoning-action-observation principle [96]. The actions performed through tool use are one of the key components that enable agent autonomy. In the context of code translation and validation, the tools can help agents explore and analyze the codebase, determine translation units, and validate translations. Existing agentic code translation techniques either employ basic, naive tools to only explore the codebase [109] or use PL-specific tools [111].

**(2) Hallucination in Repository-level Code Translation and Validation.** Translating large-scale repositories with tens or hundreds of files, specifically when translation and validation are integrated, is a long-horizon task [250], [251], [252], with hallucination being the main challenge for agents in this task. In the context of code translation, this includes hallucinating about class/file/method/variable names, finding matching libraries, or translating test assertions. Without specific consideration of trajectory context and of hallucinations, an agentic code translation and validation may generate code that does not compile or preserve the original functionality.

**(3) Dichotomy of Test Translation.** Translating first and validating next approach

[19] C2Rust [18] (100K+ LoC).

[20] AlphaTrans [45] ($11K$ LoC), Oxidizer [76] ($19K$ LoC), and Skel [77] ($4K$ LoC).

simply does not work for large-scale repository-level translation because of long call chains and test coupling effect [45]. Such systems mostly use existing developer-written tests to validate functional equivalence, either through test translation and execution or language interoperability. Given that language interoperability may not exist for arbitrary PL pairs, a PL-agnostic approach may operate on test translation (of existing tests) and additional test generation. Code generation and validation, in general, are two conflicting objectives that should not be performed by one agent [115], [116], [117], [118], [119], [120]; Otherwise, the agent may modify the test rather than incorrect code translation to success, e.g., by removing or relaxing assertions. At the same time, test translation in a real-world setting is known to be even more challenging than code translation [22], [92], requiring the code as context to understand the structure of complex objects. As a result, a naive separation of agents, one for translation and one for validation, may not work.

(4) **Transparent and Process-centric Evaluation.** Existing agentic techniques rarely discuss the principled design space of an agentic code translation workflow [109], [110], [111]. They do not evaluate how architectural choices influence the final translation quality. Although they all validate translations through test execution, there's a dearth of discussion on the limitations of test translation [45], e.g., deletion of assert statements during test translation, generation of new tests with none to low-quality assertions, and threats to the validity of findings due to corresponding false positives [50]. There is also a dearth of transparency in cost reporting, and no attempts to analyze trajectories beyond final translation outcomes.

This paper presents ReCodeAgent, a multi-agent framework for *language-agnostic*, repository-level code translation and validation (§7.3). ReCodeAgent leverages four specialized agents, dividing the overall task into distinct phases: analysis, planning, translation, and validation. The Analyzer Agent (§7.3.2) explores the source project to create a high-level translation design and determine idiomatic alternatives of third-party libraries in the target PL using Model Context Protocol (MCP) tools to the agent. The Planning Agent (§7.3.3) identifies translation units, constructs a concrete project skeleton, and devises a plan with specific steps for subsequent agents. This agent specifically addresses the complex engineering effort required by existing neuro-symbolic techniques—*challenge 1*—by replacing their PL-dependent program analysis component (e.g., dependency graph construction using CodeQL [181]) with tool-assisted, LLM-centered static analysis (lightweight tools that support many PLs, e.g., Tree-sitter [113]). The plan guides subsequent agents with concrete steps to avoid unnecessary exploration, which can cause hallucinations (*challenge 2*). The Translator agent (§7.3.4) carries out the steps outlined in the plan to co-translate code and tests. The Validator agent (§7.3.5) executes the translated tests or generates new tests to validate the translation. Separating the dynamic validation workflow from the Validator agent, while the Translator agent translates both code and test, addresses *challenge 3*. Translating tests is essential for the pipeline's generalizability beyond command-line tools, compared with Guan et al. [110] and Li et al. [111].

We evaluate the effectiveness of ReCodeAgent against four existing neuro-symbolic and agentic techniques [45], [76], [77], [109]. Our benchmark comprises 4,583 translation units, drawn from 118 real-world projects totaling over 230$K$ LoC (§7.4.1). Prior techniques translate these projects between four PL pairs, namely C-Rust, Go-Rust, Java-Python, and Python-JavaScript. On average, the translated projects by ReCodeAgent are 99.4% and 86.5% correct in terms of compilation success and test pass rate, 2.5% and 60.8% higher than those of alternative approaches (§7.4.2). When translating tests, ReCodeAgent achieves 99.3%, 0.91, and 94.9% assertion equivalence, cosine similarity, and assertion type match, respectively, demonstrating their quality (§7.4.3). Ablation study shows that removing the Analyzer, Planning, and Validator agents reduces the test pass rate by 22.7%, 25.3%, and 30.3%, respectively, while increasing trajectory complexity by 28%, measured by two process-centric metrics [39]. Comparison with two baseline agents indicates that they significantly underperform RECODEAGENT, achieving only 25.3% (↓61.2%) and 24.1% (↓62.4%) test pass rate (§7.4.4). RECODEAGENT is cost-effective, translating and validating projects in 57 minutes with the cost of $15.3, on average (§7.4.5). These results confirm that RECODEAGENT is a viable alternative to prior approaches and is vastly easier to adapt to new PL pairs. Our contributions are:

**(1)** ***Technique.*** RECODEAGENT is the first multi-agent, PL-agnostic pipeline for repository-level code translation and validation. It does not require major engineering effort or dependency on external tools.

**(2)** ***Empirical Evaluation.*** We rigorously evaluate RECODEAGENT on 118 real-world repository-level projects and 4 PL pairs against the state-of-the-art neuro-symbolic and agentic techniques. The results indicate that RECODEAGENT outperforms existing techniques in repository-level code translation and validation, without the need for PL-specific engineering effort.

**(3)** ***Tool.*** The implementation of RECODEAGENT, the agent logs and trajectories required to reproduce the results presented in this paper are publicly available [43].

## 7.2 Problem Definition and Architecture Design

A source project $P_s = (F_s, T_s, D_s)$ consists of source functions $F_s = \{f_s^1, \dots, f_s^n\}$, tests $T_s$, and dependencies $D_s$, written in language $L_s$. Given a target language $L_t$, the *repository-level code translation problem* is to produce $P_t = (F_t, T_t, D_t)$ such that: (1) $P_t$ compiles without errors, (2) $\forall i, \forall \vec{x}_s, \forall \vec{x}_t : \vec{x}_s \mapsto \vec{x}_t \implies [\![f_s^i]\!](\vec{x}_s) \simeq [\![f_t^i]\!](\vec{x}_t)$, i.e., corresponding functions are semantically equivalent, and (3) all translated tests $T_t$ pass. Here, $\vec{x}_s$ and $\vec{x}_t$ denote function inputs in languages $L_s$ and $L_t$, respectively, $\mapsto$ is a

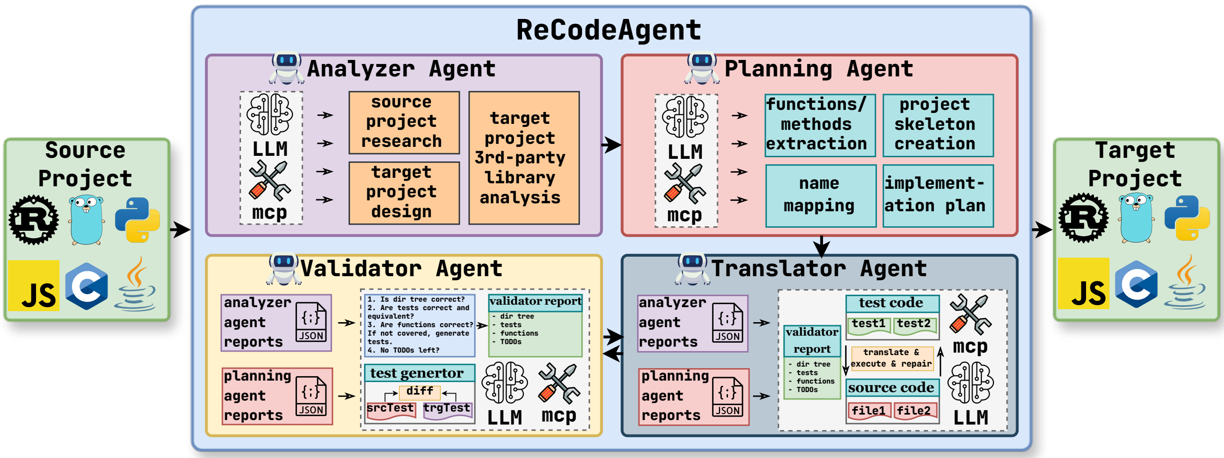



Figure 22: Overview of ReCodeAgent.

mapping of concrete values in $L_s$ to $L_t$, $[\![\cdot]\!]$ denotes semantic interpretation, and $\simeq$ denotes observational equivalence. However, given the practical limitations of current testing and verification techniques, in practice we only consider a subset of all inputs to each $f_s^i$ and $f_t^i$.

Figure 22 presents an overview of ReCodeAgent, which takes $P_s$ and $L_t$ as input and produces $P_t$. It consists of four components: the *Analyzer Agent* (§7.3.2), *Planning Agent* (§7.3.3), *Translator Agent* (§7.3.4), and *Validator Agent* (§7.3.5). The first two analyze $P_s$ and generate a dependency-aware implementation plan, while the last two execute an iterative translate–validate–repair loop to produce $P_t$.

The *Analyzer Agent* performs extensive analysis of $P_s$. It analyzes the codebase and produces a report summarizing project structure, data models, classes, interfaces, structs, error-handling strategy, and $D_s$. It then analyzes library usage in $L_s$ by consulting documentation and identifying suitable counterparts in $L_t$. This component concludes by producing a target project design document that specifies how modules should be translated and which libraries should be used to preserve functionality.

The *Planning Agent* decomposes the translation task into concrete sub-tasks: it identifies all functions in $F_s$ that require translation and constructs a consistent name

Table 20: Details of MCP tools used in RECODEAGENT.

| Server | API | Description |
|---|---|---|
| LSP | `definition(symbol_name: str)` | Read the source code definition of a symbol (function, type, constant, etc.) from the codebase. Returns the complete implementation code where the symbol is defined. |
| | `diagnostics(file_path: str)` | Get diagnostic information for a specific file from the language server. The diagnostics report contains potential issues and warnings from the language server. |
| | `edit_file(edits: List, file_path: str)` | Apply multiple text edits to a file. It takes a list of edits and a file path as input and applies all edits at once. |
| | `hover(file_path: str, line: int, column: int)` | Get hover information (type, documentation) for a symbol at the specified position. |
| | `references(symbol_name: str)` | Find all usages and references of a symbol throughout the codebase. Returns a list of all files and locations where the symbol appears. |
| | `rename_symbol(file_path: str, line: int, column: int, new_name: str)` | Rename a symbol (variable, function, class, etc.) at the specified position and update all references throughout the codebase. |
| PA | `get_directory_tree(path: str, print_dirs_only: bool)` | Traverses the specified directory and creates a hierarchical tree view using ASCII characters. |
| | `get_file_structure(language: str, file_path: str)` | Analyzes source code files and creates a structured representation of their contents, identifying key elements like classes, functions, methods, imports, and global variables. |

mapping to ensure uniformity in $P_t$. It also generates a skeleton structure for $P_t$, outlining file organization and module boundaries, and produces an implementation plan with concrete tasks for translating and validating $P_s$.

The *Translator Agent* and *Validator Agent* execute the implementation plan. The *Translator Agent* translates $F_s$ and $T_s$ into $F_t$ and $T_t$, incrementally filling in the skeleton files; if validation fails, it uses the Validator Agent's report to repair translation bugs. The *Validator Agent* independently validates $P_t$ by executing $T_t$ and performing coverage-gap analysis; when functions in $F_t$ are uncovered, it triggers additional test generation and reports results back to the Translator Agent for repair.

### 7.3 RECODEAGENT

Tools are essential and key components to enable autonomy for agents. In this section, we first explain the tools that assist RECODEAGENT agents with static analysis (§7.3.1) and explain the details and workflow of each agent for code translation and validation through reasoning and tool usage (§7.3.2–§7.3.5).

#### 7.3.1 Tools

RECODEAGENT leverages a set of Model Context Protocol (MCP) tools to provide static analysis capabilities and facilitate effective agent interactions with the codebase. The

MCP is an open-source protocol that allows LLMs to seamlessly connect to external data, tools, and software systems. We implement our custom tools and expose them to each agent in ReCodeAgent through MCP. These tools enable agents to obtain detailed project information, modify code, and retrieve documentation in a PL-agnostic manner. Table 20 summarizes the main MCP tools and their description, which are integrated into ReCodeAgent:

***Language Server Protocol (LSP) Tools:*** LSP is an open, JSON-RPC-based protocol for use between source code editors or integrated development environments (IDEs) and servers that provide language intelligence: PL-specific features like code completion, syntax highlighting and marking of warnings and errors, as well as refactoring routines [114]. The goal of the protocol is to allow PL support to be implemented and distributed independently of any given IDE. We use LSPs from six PLs [253], [254], [255], [256], [257], [258] as a set of tools that allow agents to interact with the codebase at a semantic level, independent of the underlying PL. Extending support to more PLs only requires installing their language server which is usually maintained and available online [259], without writing any additional code. LSP functionalities that we implement include:

1. `definition`: It takes a symbol name as input and retrieves the complete implementation (e.g., function, class) along with the file path and line numbers from the codebase, enabling agents to understand and extract source code fragments for translation.

2. `diagnostics`: Provides diagnostic information such as errors and warnings for a specified file, which assists agents in identifying potential issues in both source and translated code. IDEs usually indicate errors and warnings using red and yellow underlines in the code editor.

3. `edit_file`: Applies a set of text edits to a file atomically, supporting incremental

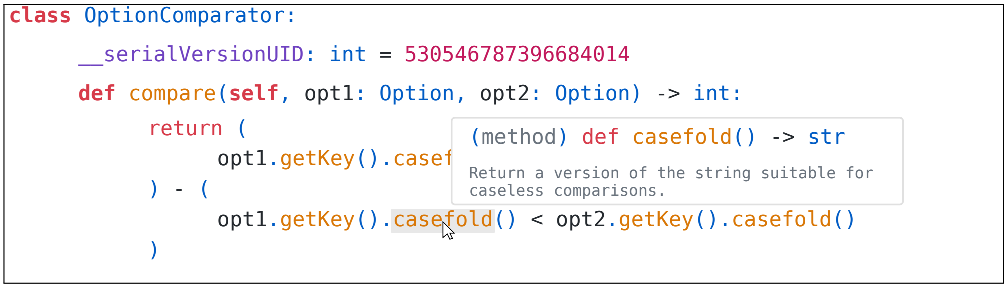



Figure 23: Hover feature on a Python code in an IDE.

```
==================================================== CALL ==================================================
hover(MCP)(file_path: "OptionComp.py", line: 7, column: 27)
================================================== RESPONSE ================================================
```python
(method) def casefold() -> str
'''
Return a version of the string suitable for caseless comparisons.
```

Figure 24: Hover tool from the Python LSP server.

construction and refinement of the translated project. This tool is helpful when there are a large number of edits that need to be applied, as opposed to using the agent's `Edit` tool once per every edit.

4. **`hover`**: Returns documentation for a symbol at a specified position in the code, including docstrings and code deprecation details. Figure 23 shows the hover feature in a normal IDE, while the same functionality from the LSP server is given in Figure 24.

5. **`references`**: Locates all occurrences and usages of a symbol across the codebase, essential for large-scale code refactoring by the agent. This tool returns exact line numbers of every usage, making it easy for agents to refer to them later.

6. **`rename_symbol`**: Renames a symbol at a given location and updates all corresponding references throughout the project. This tool is helpful in making consistent changes across the codebase, without the need to perform additional edits.

***Project Analysis (PA) Tools:*** These tools extract structural information from the
```

```
=============== get_directory_tree ===============
|-- python/
  |-- src/
  |  |-- main/
  |  |  |-- __init__.py
  |  |  |-- BasicParser.py
  |  |-- test/
  |  |  |-- __init__.py
  |  |  |-- BasicParserTest.py
  |-- conftest.py
  |-- pytest.ini
  |-- run.sh
```

```
=============== get_file_structure ===============
{
 "filepath": "/../../../"
 "language": "java/python/..."
 "skeleton": {
   "imports": [...]
   "classes": [...]
   "functions": [...]
   "globals": [...]
   "structs": [...]
 }
}
```

Figure 25: Project analysis (PA) tools output.

```
==== Source Project Research ===
## Overview
## Directory Structure
## Structs and Interfaces
## Data Models
## Error Handling
## Dependencies
```

```
= Third-Party Library Analysis =
## Library A
  → Overview
  → Usages
  → Example
  → Recommendations in Target PL
## Library B
```

```
===== Target Project Design ====
## Overview
## Translation Requirements
## Source Files to Translate
## Module Structure
## Error Handling
## Third-Party Libraries
```

Figure 26: Documents generated by *Analyzer Agent* in RECODEAGENT.

codebase, aiding agents in project comprehension and planning. The goal of these tools is to reduce the token consumption of the agent, which would otherwise be spent on exploring the codebase and files. Figure 25 shows two sample outputs from project analysis tools. Functionalities included are:

1. `get_directory_tree`: Returns a structured representation of the project directory, printing files such as main, test, and configuration and their directory hierarchy.

2. `get_file_structure`: Generates a structured representation of a given source file, identifying key code elements such as classes, functions, structs, and global variables.

### 7.3.2 Analyzer Agent

The *Analyzer Agent* conducts initial research and formulates the high-level design of the translation (Algorithm 6, line 28). This agent ensures that the target project preserves structurally similar to the source, and identifies the most suitable libraries and design

**Algorithm 6** RECODEAGENT

**Require:** *sourceProject*, *llm*, *tools*, *timeout* $= 5000s$, *maxIter* $= 5$, *validationReport* $\leftarrow \emptyset$
**Ensure:** *translatedValidatedProject*

```
procedure TranslatorAgent(context, validationReport)
    translatorAgent ← initializeAgent(llm, tools, context)
    if validationReport ≠ ∅ then
        return translatorAgent.repair(validationReport)
    end if
    implementationPlan ← context.planningOutput.plan
    translatedProject ← ∅
    for all Part-A ∈ implementationPlan do
        agentTranslatedSourceCode ← translatorAgent.implement(Part-A)
        translatedProject ← translatedProject ∪ agentTranslatedSourceCode
    end for
    for all Part-B ∈ implementationPlan do
        agentTranslatedTestCode ← translatorAgent.implement(Part-B)
        translatedProject ← translatedProject ∪ agentTranslatedTestCode
    end for
    return translatedProject
end procedure

procedure ValidatorAgent(context, translatedProject)
    validatorAgent ← initializeAgent(llm, tools, context)
    validationReport ← validatorAgent.validateTranslations(translatedProject)
    if validationReport.hasUncoveredFunctions() then
        agentGeneratedTestCode ← validatorAgent.generateAndValidateTests()
        translatedProject ← translatedProject ∪ agentGeneratedTestCode
    end if
    translatedValidatedProject ← translatedProject
    return translatedValidatedProject, validationReport
end procedure

analyzerOutput ← runAnalyzerAgent(sourceProject)                          ▷ if !timeout
planningOutput ← runPlanningAgent(sourceProject, analyzerOutput)          ▷ if !timeout
context ← sourceProject ∪ analyzerOutput ∪ planningOutput
for iteration ← 1 to maxIter do
    translatedProject ← TranslatorAgent(context, validationReport)        ▷ if !timeout
    translatedValidatedProject, validationReport ← ValidatorAgent(context, translatedProject)    ▷ if !timeout
    if validationReport.isAllSuccess() then
        break
    end if
end for
return translatedValidatedProject
```

patterns for the target PL. Figure 26 illustrates the three documents produced by this agent, corresponding to the following phases:

***Source Project Research:*** The analyzer agent first explores the source codebase (e.g., using the `Read` tool) to ascertain its architectural design and functional requirements. Subsequently, it invokes the `get_directory_tree` tool to extract the project's directory structure, which serves as the foundational blueprint for translation. To get a semantic understanding of the codebase, the agent employs the `get_file_structure` and LSP tools to analyze the contents of each file in greater detail. The output of this phase is a research document that includes the source project's dependencies, error handling mechanisms, and directory hierarchy.

***Third-Party Library Analysis:*** Next, the agent identifies all third-party and standard libraries utilized within the source project. For each identified dependency, the agent investigates idiomatic counterparts available in the target PL. Leveraging the `WebFetch` and `hover` tools, the agent retrieves official documentation to determine recommended usage patterns and evaluate the trade-offs associated with alternative library selections. These findings are consolidated into a document, ensuring that subsequent translation phases are guided by current best practices within the target PL ecosystem.

***Target Project Design:*** In the final phase, the agent synthesizes its research into a comprehensive target project design document, which enforces a strict one-to-one structural mapping between source and target projects, covering directory structure, file organization, and identifier naming conventions (classes, methods, and variables). The document specifies which files require translation, how source constructs map to target equivalents (e.g., Java Interfaces → Rust Traits, Go Structs → Python Classes), and outlines strategies for error handling and library integration. This document serves as the authoritative reference for the subsequent *Planning* (§7.3.3), *Translator* (§7.3.4), and *Validator* (§7.3.5) *Agents.*

```
= Fragment Extraction =
## cd.go
cd.go:isNum
cd.go:NewLuhn
cd.go:NewDamm
cd.go:NewUP

## damm.go
```

```
===== Name Mapping ====
## functions:
go.isNum: rs.isNum
go.NewLuhn: rs.NewLuhn
go.NewDamm: rs.NewDamm
go.NewUP: rs.NewUP
## variables:
go. ...
```

```
= Skeleton Generation =
fn isNum(n: char) ->
bool {}
fn NewLuhn() -> impl P
{}
fn NewDamm() -> impl P
{}
fn NewUP() -> impl P {}
```

```
= Implementation Plan =
## Overview
## Part A:
A1: Translate cd.go
A2: Translate damm.go
## Part B:
B1: Translate cd_t.go
B2: Translate damm_t.go
```

Figure 27: Documents generated by *Planning Agent* in RECODEAGENT.

### 7.3.3 Planning Agent

The *Planning Agent* reads the source project research and target project design documents generated by the *Analyzer Agent* (§7.3.2) and decomposes the high-level design into granular, executable implementation steps (Algorithm 6, line 29). This agent ensures that every source code file is translated and validated according to a logical, dependency-aware order. Figure 27 illustrates the documents and skeleton files produced by the planning agent.

***Fragment Extraction:*** The agent first extracts all translation units—including functions, methods, and classes—from both source and test files. Fragment extraction is performed using the `get_file_structure` tool, while maintaining a strict validation-in-the-loop process. For validation, the agent generates executable scripts to verify that every extracted fragment exists in the source codebase and that no files have been omitted. This validation step mitigates agent hallucination, wherein the agent erroneously concludes that a task has been completed when it has not. This phase concludes by generating a document of extracted fragments, with each fragment recorded in the format `file_name:fragment_name`.

***Name Mapping:*** To ensure one-to-one translation and naming consistency, the agent constructs a mapping from source fragments to their target counterparts, strictly preserving symbol names (e.g., `camelCase`, `snake_case`) to maintain functional parity across the project. This mapping is then used during skeleton generation to produce accurate method signatures in the target PL, preventing LLMs from arbitrarily renaming methods and classes in ways that impede translation tracking.

***Skeleton Generation:*** The agent subsequently constructs the target project's directory structure and populates it with skeleton files. These skeleton files contain class declarations and method signatures without concrete implementations. This approach provides a compilable framework that mirrors the source project's architecture, enabling the translation process to proceed incrementally.

***Implementation Plan:*** The implementation plan is a structured document partitioned into source code translation (`Part A`) and test code translation and validation (`Part B`). The plan adheres to a bottom-up ordering, ensuring that dependencies are implemented prior to the modules that rely upon them. For example, a sample step in `Part A` can be `"Translate HelpFormatter.py and validate its syntactical correctness"`. Each step in the plan is expected to yield compilable code and provides an explicit checklist for the *Translator* (§7.3.4) and *Validator* (§7.3.5) *Agents* to execute.

### 7.3.4 Translator Agent

The *Translator Agent* carries out the implementation plan by executing both `Part A` (source code translation) and `Part B` (test translation) (Algorithm 6, lines 1–17). The objective of this agent is to translate the source project into the target PL while preserving functional equivalence and architectural alignment. If the *Validator Agent* reports failures, the Translator Agent enters repair mode and applies targeted fixes based on the validation report. The agent follows a systematic workflow to ensure a one-to-one translation:

1. *Context Integration:* The agent loads the implementation plan, the target design document, and the name mapping files. This ensures that translated identifiers (e.g., class and variable names) remain consistent with the plan and are not arbitrarily renamed.

2. *Incremental Implementation:* Following the dependency-aware ordering from the

planning phase, the agent replaces stubs in the target skeleton files with complete implementations for `Part A`. It then translates developer-written tests for `Part B`, creating the corresponding test files in the target project.

3. *Language-Specific Adaptation:* When translating between languages with divergent feature sets, the agent applies targeted adaptation strategies that preserve behavior (e.g., emulating overloading via default arguments or dispatch).

4. *Repair Mode:* When provided with a non-empty validation report, the agent diagnoses the reported failures and updates the translated source and/or test code accordingly, iterating until validation succeeds or the iteration budget is exhausted.

The output of this agent is a translated project that includes both translated source code and tests, ready for the *Validator Agent*.

### 7.3.5 Validator Agent

The *Validator Agent* validates the functional correctness of the translated project (Algorithm 6, lines 18–27). Given a translated project (including translated tests) produced by the *Translator Agent*, this agent executes tests, performs coverage-gap analysis, and produces a validation report that is fed back to the *Translator Agent* for repair in the next iteration.

***Validation and Failure Reporting:*** The agent executes the translated test suite in the target environment and checks whether all tests pass. If failures occur (e.g., compilation errors or assertion failures), the agent consolidates diagnostics—including stack traces and failing test cases—into a structured validation report. This report identifies the failing functions and provides actionable feedback for the Translator Agent's repair step in the next iteration.

***Coverage-Guided Test Generation:*** To fully validate the translated modules, the agent performs a coverage-gap analysis by comparing the executed tests against the complete list of functions identified during the planning phase. If uncovered functions remain, it generates additional tests in both the source and target PLs to exercise the uncovered functions and adds them to the translated project. The generated tests are executed in both PLs to ensure matching behavior. The agent then re-executes validation to update the validation report, ensuring that the translated code is both functionally correct (with respect to the available tests) and more rigorously exercised. The iterative loop continues until all tests pass or the maximum iteration limit is reached.

### 7.4 Evaluation

To evaluate different aspects of ReCodeAgent, we investigate the following research questions:

**RQ1:** *Effectiveness of ReCodeAgent.* To what extent can ReCodeAgent effectively translate real-world projects? Can it outperform expensively developed techniques?

**RQ2:** *Test Translation.* To what degree are the translated tests equivalent to the original tests? What are the limitations of ReCodeAgent when translating tests?

**RQ3:** *Ablation Study.* To what extent do the Analyzer, Planning, and Validator agents impact the performance of ReCodeAgent? Can a standalone LLM agent perform similarly to ReCodeAgent?

**RQ4:** *Cost and Tool Usage Analysis.* How much does it cost and how long does it take for ReCodeAgent to translate projects? What kinds of tools are frequently invoked by ReCodeAgent?

#### 7.4.1 Experimental Setup

**Benchmark.** We assess the performance of ReCodeAgent using benchmarks from previously published studies on automated repository-level code translation and validation. Each benchmark contains a project implemented in a specific PL that includes both source and test code. The goal for each benchmark is to translate and validate the project in a target PL, ensuring that all tests are successfully executed and pass. Table 21 provides an overview of our open-source subject translation projects from four recent repository-level code translation techniques [45], [75], [76], [109] covering the following PLs: C, Go, Rust, Java, Python, and JavaScript. In total, our evaluation includes 118 projects spanning over $230K$ lines of code. We exclude SYZYGY [75] from our evaluation due to the unavailability of its artifact. Since the test suites of RepoTransBench [260] are written by LLMs and not validated as correct by humans, we exclude them as well. For ALPHATRANS [45], we select a subset of projects for which the authors have provided validated test suites. Moreover, the CRUST benchmark consists of 100 independent C projects translated to Rust. All these prior works produced translations of real-world open-source GitHub repositories and assessed functional equivalence via test execution.

**LLM.** RECODEAGENT works with different LLMs. Major software engineering leaderboards [232] have shown that the Claude Sonnet performs similarly to or in some cases outperforms other state-of-the-art proprietary LLMs, such as OpenAI GPT-5 and Google Gemini Pro. Therefore, we use Anthropic's `Claude 4.5 Sonnet` as the main LLM in all our experiments. To make our results reproducible without re-running experiments, RECODEAGENT logs the inputs, intermediate agent interactions, tool execution results, and outputs of the LLM, and supports replaying these logs. Each agent in RECODEAGENT terminates within the budget of 5,000 seconds, empirically set after analyzing the runtime of our largest project.

**Competing Techniques.** We compare RECODEAGENT against SKEL [77], OXIDIZER [76], ALPHATRANS [45], and SWE-agent [261] from CRUST [109]. While we cannot directly compare to ACToR [111] as its implementation is tied to CLI programs, our ablation that removes the analyzer and planner agents closely resembles its agent architecture, and can serve as a proxy for its performance[21].

[21]Confirmed by the authors of ACToR [111].

Table 21: Effectiveness of ReCodeAgent in repository-level code translation and validation in terms of test and function validation. **LoC:** Lines of Code, **CS:** Compilation Success, **TE:** # Tests Executed, **TP:** # Tests Passing, **TF:** # Tests Failing, **CRUST-**$\{\alpha, \beta, \sigma, \gamma\}$: $\alpha$: both compile, $\beta$: only ReCodeAgent compile, $\sigma$: only SWE-agent compile, $\gamma$: both do not compile, C: Test coverage, C+: Increase in test coverage. Tuple entries indicate ⟨Tool, ReCodeAgent ⟩.

| Tool (PL Pair) | Project | LoC | CS (%) | # Validated Developer Tests | Validated Developer Tests | | | ReCodeAgent Translated Developer Tests | | | ReCodeAgent Generated Tests | | | | | Function Validation | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | TE | TP | TF | TE | TP | TF | TE | TP | TF | C (%) | C+ (%) | Total | Success | Fail |
| Oxidizer [76] (Go→Rust) | checkdigit [210] | 428 | ⟨100, 100⟩ | 36 | ⟨36, 36⟩ | ⟨33, 36⟩ | ⟨3, 0⟩ | 36 | 36 | 0 | 71 | 71 | 0 | 79.7 | 94.7 | 29 | ⟨21, 29⟩ | ⟨8, 0⟩ |
| | go-edlib [211] | 639 | ⟨100, 100⟩ | 36 | ⟨36, 36⟩ | ⟨19, 36⟩ | ⟨17, 0⟩ | 36 | 36 | 0 | 3 | 3 | 0 | 94.7 | 94.9 | 24 | ⟨18, 24⟩ | ⟨6, 0⟩ |
| | histogram [212] | 314 | ⟨100, 100⟩ | 2 | ⟨2, 2⟩ | ⟨2, 2⟩ | ⟨0, 0⟩ | 2 | 2 | 0 | 66 | 66 | 0 | 38.0 | 90.5 | 19 | ⟨12, 19⟩ | ⟨7, 0⟩ |
| | nameparts [213] | 413 | ⟨100, 100⟩ | 26 | ⟨26, 26⟩ | ⟨23, 26⟩ | ⟨3, 0⟩ | 26 | 26 | 0 | 22 | 22 | 0 | 96.8 | 96.8 | 15 | ⟨9, 14⟩ | ⟨6, 1⟩ |
| | stats [214] | 1241 | ⟨100, 100⟩ | 121 | ⟨121, 121⟩ | ⟨71, 121⟩ | ⟨50, 0⟩ | 121 | 121 | 0 | 320 | 320 | 0 | 43.3 | 79.6 | 52 | ⟨38, 52⟩ | ⟨14, 0⟩ |
| | textrank [215] | 1132 | ⟨100, 100⟩ | 8 | ⟨8, 8⟩ | ⟨6, 8⟩ | ⟨2, 0⟩ | 8 | 8 | 0 | 127 | 127 | 0 | 72.6 | 98.7 | 52 | ⟨40, 52⟩ | ⟨12, 0⟩ |
| **Total** | | 4167 | ⟨100, 100⟩ | 229 | ⟨229, 229⟩ | ⟨154, 229⟩ | ⟨75, 0⟩ | 229 | 229 | 0 | 609 | 609 | 0 | 70.9 | 92.5 | 191 | ⟨138, 190⟩ | ⟨53, 1⟩ |
| AlphaTrans [45] (Java→Python) | cli [186] | 37841 | ⟨100, 100⟩ | 381 | ⟨66, 381⟩ | ⟨35, 360⟩ | ⟨31, 21⟩ | 381 | 381 | 0 | 257 | 257 | 0 | 96.7 | 97.3 | 257 | ⟨196, 241⟩ | ⟨61, 16⟩ |
| | csv [188] | 33072 | ⟨100, 100⟩ | 298 | ⟨147, 298⟩ | ⟨3, 241⟩ | ⟨144, 57⟩ | 298 | 298 | 0 | 192 | 190 | 2 | 84.4 | 85.8 | 213 | ⟨74, 211⟩ | ⟨139, 2⟩ |
| | fileupload [191] | 3567 | ⟨100, 100⟩ | 39 | ⟨39, 39⟩ | ⟨36, 39⟩ | ⟨3, 0⟩ | 39 | 39 | 0 | 208 | 208 | 0 | 38.7 | 71.6 | 25 | ⟨19, 25⟩ | ⟨6, 0⟩ |
| | validator [195] | 41605 | ⟨100, 100⟩ | 463 | ⟨359, 463⟩ | ⟨114, 438⟩ | ⟨245, 25⟩ | 463 | 435 | 28 | 132 | 131 | 1 | 65.0 | 73.0 | 409 | ⟨217, 397⟩ | ⟨192, 12⟩ |
| **Total** | | 116085 | ⟨100, 100⟩ | 1181 | ⟨611, 1181⟩ | ⟨188, 1078⟩ | ⟨423, 103⟩ | 1181 | 1153 | 28 | 789 | 786 | 3 | 71.2 | 81.9 | 904 | ⟨506, 874⟩ | ⟨398, 30⟩ |
| Skel [77] (Python→JavaScript) | bst [216] | 123 | ⟨100, 100⟩ | 11 | ⟨11, 11⟩ | ⟨11, 11⟩ | ⟨0, 0⟩ | 11 | 11 | 0 | 6 | 6 | 0 | 89.7 | 99.0 | 21 | ⟨21, 21⟩ | ⟨0, 0⟩ |
| | colorsys [217] | 120 | ⟨100, 100⟩ | 2 | ⟨2, 2⟩ | ⟨2, 2⟩ | ⟨0, 0⟩ | 2 | 2 | 0 | 46 | 46 | 0 | 87.0 | 91.3 | 9 | ⟨9, 9⟩ | ⟨0, 0⟩ |
| | heapq [218] | 189 | ⟨100, 100⟩ | 8 | ⟨8, 8⟩ | ⟨7, 8⟩ | ⟨1, 0⟩ | 8 | 8 | 0 | 11 | 11 | 0 | 91.6 | 91.6 | 24 | ⟨23, 24⟩ | ⟨1, 0⟩ |
| | html [219] | 684 | ⟨100, 100⟩ | 7 | ⟨7, 7⟩ | ⟨6, 7⟩ | ⟨1, 0⟩ | 7 | 7 | 0 | 13 | 13 | 0 | 77.1 | 86.1 | 42 | ⟨39, 42⟩ | ⟨3, 0⟩ |
| | mathgen [220] | 735 | ⟨100, 100⟩ | 5 | ⟨5, 5⟩ | ⟨4, 5⟩ | ⟨1, 0⟩ | 5 | 5 | 0 | 11 | 11 | 0 | 96.4 | 98.6 | 82 | ⟨79, 82⟩ | ⟨3, 0⟩ |
| | rbt [221] | 366 | ⟨100, 100⟩ | 10 | ⟨10, 10⟩ | ⟨10, 10⟩ | ⟨0, 0⟩ | 10 | 10 | 0 | 5 | 5 | 0 | 87.1 | 88.4 | 27 | ⟨27, 27⟩ | ⟨0, 0⟩ |
| | strsim [222] | 654 | ⟨100, 100⟩ | 19 | ⟨19, 19⟩ | ⟨19, 19⟩ | ⟨0, 0⟩ | 19 | 19 | 0 | 64 | 64 | 0 | 88.8 | 94.1 | 50 | ⟨50, 50⟩ | ⟨0, 0⟩ |
| | toml [223] | 1206 | ⟨100, 100⟩ | 12 | ⟨12, 12⟩ | ⟨10, 12⟩ | ⟨2, 0⟩ | 12 | 12 | 0 | 150 | 150 | 0 | 72.6 | 83.2 | 47 | ⟨43, 47⟩ | ⟨4, 0⟩ |
| **Total** | | 4077 | ⟨100, 100⟩ | 74 | ⟨74, 74⟩ | ⟨69, 74⟩ | ⟨5, 0⟩ | 74 | 74 | 0 | 306 | 306 | 0 | 86.3 | 91.5 | 302 | ⟨291, 302⟩ | ⟨11, 0⟩ |
| SWE-agent [100] (C→Rust) | Crust-$\alpha$ [109] | 22961 | ⟨40, 40⟩ | 166 | ⟨153, 166⟩ | ⟨130, 146⟩ | ⟨23, 20⟩ | - | - | - | 493 | 493 | 0 | 68.2 | 75.7 | 673 | - | - |
| | Crust-$\beta$ [109] | 66704 | ⟨0, 49⟩ | 321 | ⟨0, 320⟩ | ⟨0, 295⟩ | ⟨0, 25⟩ | - | - | - | 1118 | 1114 | 4 | 57.9 | 75.2 | 1900 | - | - |
| | Crust-$\sigma$ [109] | 3894 | ⟨1, 0⟩ | 1 | ⟨1, 0⟩ | ⟨1, 0⟩ | ⟨0, 0⟩ | - | - | - | 53 | 51 | 2 | 4.3 | 67.4 | 41 | - | - |
| | Crust-$\gamma$ [109] | 15169 | ⟨0, 0⟩ | 135 | ⟨0, 0⟩ | ⟨0, 0⟩ | ⟨0, 0⟩ | - | - | - | 274 | 270 | 4 | 27.0 | 44.7 | 572 | - | - |
| **Total** | | 108728 | ⟨41, 89⟩ | 623 | ⟨154, 486⟩ | ⟨131, 441⟩ | ⟨23, 45⟩ | - | - | - | 1938 | 1928 | 10 | 39.3 | 65.7 | 3186 | - | - |
| **Total** | | 233057 | ⟨96.9, 99.4⟩ | 2107 | ⟨1068, 1970⟩ | ⟨542, 1822⟩ | ⟨526, 148⟩ | 1484 | 1456 | 28 | 3642 | 3629 | 13 | 70.8 | 85.4 | 4583 | ⟨935, 1366⟩ | ⟨462, 31⟩ |

**Implementation.** For validating translations, ReCodeAgent uses Rust 1.92.0, Python 3.12.9, Java 21.0.7, Node 22.16.0, GCC 12.2.0, and Go 1.24.4. We use Anthropic's Claude Code 2.1.19 [206] for the agentic workflow discussed in §7.3.

### 7.4.2 RQ1: Effectiveness of ReCodeAgent

Table 21 shows the results of ReCodeAgent and other techniques in repository-level code translation and validation. We assess effectiveness from three different aspects: (1) *Syntactic Correctness*, (2) *Test Validation*, and (3) *Function Validation*.

**Syntactic Correctness.** ReCodeAgent achieves an overall Compilation Success (CS) of 99.4% across all projects, surpassing competing techniques which attain 96.9%. For projects in Oxidizer, AlphaTrans, and Skel, both ReCodeAgent and existing techniques produce 100% compilable code. The most significant improvement is observed in Crust, where ReCodeAgent generates compilable translations for $\frac{89}{100}$ projects–an improvement of 48 projects over SWE-agent. This improvement is particularly notable given the difficulty of C→Rust translation, especially with respect to memory management and ownership semantics. For instance, the following function `writechar` from `printf` is translated properly by SWE-agent; however, its call sites inconsistently use `int` and `char` as the first argument. While this behavior is acceptable in C, where a `char` is represented as an `int` in memory, it is invalid in Rust, where these types are distinct and thus lead to compilation errors. In contrast, ReCodeAgent consistently invokes `writechar` with the appropriate argument type `char`.

```c
-------------------- C SOURCE CODE -------------------
int        writechar(char c, int *len) {
    return ((*len)++, write(1, &c, 1));
}
```

```rust
----------------- RUST TRANSLATION -----------------
pub fn writechar(c: char, len: &mut i32) -> i32 {
    *len += 1; ...
}
```

**Test Validation.** To evaluate the functional equivalence of translations, we execute source PL developer tests and measure the number of tests executed and passing. If existing tests do not cover certain functions, ReCodeAgent generates tests to validate them.

*Validated Developer Tests.* To fairly evaluate translations across all projects and to eliminate the threat of incorrectly translated tests, we use the validated test suites provided in the artifacts of prior tools. Because Oxidizer does not translate tests, we manually translated and validated the Go tests into Rust. Multi-column *Validated Developer Tests* in Table 21 shows the number of executed, passing, and failing tests for existing tools and ReCodeAgent. As corroborated in the table, ReCodeAgent substantially improves test pass rate (TPR), passing $\frac{1{,}822}{2{,}107}$ tests (86.5%), compared to only $\frac{542}{2{,}107}$ tests (25.7%) for competing techniques, improving TPR by 60.8%. In particular, ReCodeAgent achieves 100% TPR compared to Oxidizer and Skel which achieve 67.2% and 93.2%, respectively. For the Crust benchmark, our comparison is restricted to the 40 projects for which both ReCodeAgent and SWE-agent produce compilable translations (Crust-$\alpha$). Out of 166 available tests, ReCodeAgent executes and passes 146 tests (88.0% TPR), while SWE-agent achieves a TPR of 78.3%. The largest gain is observed in AlphaTrans, where ReCodeAgent passes 1,078 tests compared to only 188 tests by AlphaTrans's compositional approach, an improvement of 75.4%. The reduced performance of AlphaTrans is primarily due to its limited number of executed tests caused by test collection errors; for example, in `cli` only $\frac{66}{381}$ tests are executed. The following snippet illustrates one such problematic translation that is required by most test classes and leads to test collection errors. The Java code uses a `protected` constructor to restrict who can create `CommandLine` instances while still allowing controlled construction within the package or subclasses. By contrast, the Python translation replaces this with a runtime check that always raises a `TypeError` when `CommandLine` is instantiated directly, effectively making the class non-instantiable and changing the original design intent.

```java
------------------ JAVA SOURCE CODE ------------------
public class CommandLine implements Serializable {

    protected CommandLine() {}
}
```

```python
----------------- PYTHON TRANSLATION -----------------
class CommandLine:
    def __init__(self) -> None:
        if type(self) is CommandLine:
            raise TypeError("Error")
```

*ReCodeAgent Translated Developer Tests.* In addition to evaluating translations using validated developer tests, we also execute developer tests translated by ReCodeAgent to assess its capability in test translation. A detailed analysis of translated test quality is provided in §7.4.3. Multi-column *ReCodeAgent Translated Developer Tests* in Table 21 summarize these results. Across 1,484 translated tests, excluding Crust where test translation is not required, ReCodeAgent executes and passes 1,456 tests (98.1%), with only 28 failures. Except for AlphaTrans, ReCodeAgent can correctly translate and produce tests equivalent to those in the source PL in Oxidizer and Skel mostly because they have simpler test logic. We further analyzed the discrepancies in test failures between validated developer tests and translated ones. Specifically, we identified incorrectly validated tests by the authors of AlphaTrans as shown below. This example is from `csv` project with 57 test failures from validated developer tests, but none from ReCodeAgent translated tests. The `printRecord1` invocation in `testJiraCsv249` takes two string arguments in source Java tests, but was incorrectly translated to take a list in Python and therefore fails. This test translated by ReCodeAgent has the same semantics as Java and passes correctly, demonstrating its ability in automated test translation.

```
------------------ JAVA TEST CODE ------------------
public void testJiraCsv249() {
    ...
    printer.printRecord1("foo \\", "bar");
    ...
}
```

```
-------------- ALPHATRANS TRANSLATION --------------
def testJiraCsv249(self) -> None:
    ...
    printer.printRecord1(["foo \\", "bar"])
    ...
```

*ReCodeAgent Generated Tests.* A major limitation of source PL developer tests is their low coverage and the inability to validate unexercised translations. For instance, the `fileupload` project in AlphaTrans has a line coverage of only 38.7%, leaving the remaining 61.3% of translated lines unvalidated. To mitigate this, ReCodeAgent generates additional tests for each project (§7.3.5). This capability addresses a fundamental limitation in existing validation approaches: the quality of validation is inherently bounded by the coverage of the original test suite. Projects with low test coverage may have large portions of translated code that remain unvalidated, potentially hiding translation bugs. Multi-column *ReCodeAgent Generated Tests* indicates generated tests executed, passing, and failing. Across all benchmarks, ReCodeAgent generates 3,642 tests and achieves a TPR of 99.6% (3,629 passing, 13 failing). The high pass rate on generated tests indicates that ReCodeAgent's test generation component produces valid tests that correctly exercise the translated code. Moreover, the generated tests increase test coverage significantly: on average, test coverage improves from 70.8% to 85.4%, representing a 14.6% improvement. The coverage improvement is particularly valuable for the AlphaTrans benchmark, where the original projects have varying levels of test coverage. By generating additional tests, ReCodeAgent validates code paths that were previously unvalidated, increasing confidence in the correctness of the translated implementation.

**Function Validation.** So far, we only used test validation for evaluating the functional correctness of translations. However, the problem of *test translation coupling effect* discussed in AlphaTrans [45] still exists. That is, a translation issue in one method casts a shadow in validating the translation of the other methods. Consequently, test validation alone is not a good metric for evaluating functional correctness, as it can heavily favor one technique over the others. To address this, we evaluate each translated function independently as an alternative way to measure correctness.

For benchmarks where function-level validation is possible, we evaluate whether each translated function produces correct output when invoked with the same inputs as the original function. Since Crust only performs test validation, we excluded it from function validation. Multi-column *Function Validation* shows the results of this evaluation. Across 1,397 functions, ReCodeAgent achieves successful validation for 1,366 (97.8%) compared to 935 (66.9%) for competing techniques. This improvement of 30.9% demonstrates that test validation (60.8% improvement in TPR) alone can be unreliable and produce inflated improvements. For instance, let's consider the following example from `nameparts` project in Oxidizer which validates the `Parse` function. When doing test validation (left), the test only checks if the function panics on the input `"I am a Popsicle"`. However, when performing function validation (right), the test goes beyond checking for panic, and asserts the parsed properties, i.e., `FullName`. As a result, test validation validates the `Parse` function as correct since it does not panic, however, function validation fails because `FullName` is not parsed correctly, demonstrating function validation's more rigorous evaluation.

```
------------------ TEST VALIDATION ------------------
fn TestObviouslyBadName() {
    let result = std::panic::catch_unwind(|| {
        Parse("I am a Popsicle".to_string())
    });
    assert!(result.is_ok(),
    "Parse should not panic on invalid input");
}
```

```
---------------- FUNCTION VALIDATION ----------------
pub fn parse__unit_test() {
    let result = Parse(input_name);
    assert_eq!(result.FullName,
    expected_result["FullName"].as_str()
    .unwrap_or(""),
    "Test case {}: FullName mismatch", i);
}
```

The function-validation results above highlight that rigorous, per-function equivalence checking exposes bugs that repository-level test execution can miss. Chapter 6 develops such techniques for this setting: given a source function and its translation, MatchFixAgent combines approximate semantic analyses with targeted test generation and repair to produce a function-level equivalence verdict. The two systems are therefore complementary rather than redundant—ReCodeAgent addresses end-to-end repository translation and validation within a single agentic loop, while MatchFixAgent specializes in deeper validation and repair of individual source–translation pairs. A natural composition is to integrate MatchFixAgent into ReCodeAgent so that the

Table 22: Comparison between ReCodeAgent translated tests and original source PL tests. Tuple entries indicate ⟨Source Test, Translated Test⟩. LoC: Lines of Code.

| Tool (PL Pair) | Project | # Tests | # Tests Translated / Not Translated | # Tests w/ Matching / Non-Matching # Assertions | # Total / Matching assertEqual Output | Assertion Type Match (%) | | | | Avg. Cosine Similarity | Avg. LoC | Avg. # Method Invocations |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Assert Equal | Assert True | Assert False | Other | | | |
| Oxidizer (Go→Rust) | checkdigit | 36 | 36/0 | 36/0 | 45/45 | 100 | - | - | - | 0.94 | ⟨24.42, 24.58⟩ | ⟨7.14, 5.33⟩ |
| | go-edlib | 36 | 36/0 | 36/0 | 45/45 | 84.44 | - | - | - | 0.92 | ⟨30.78, 51⟩ | ⟨7.61, 9.50⟩ |
| | histogram | 2 | 2/0 | 2/0 | 11/11 | 100 | - | - | - | 0.96 | ⟨23.50, 16.50⟩ | ⟨24, 16.50⟩ |
| | nameparts | 26 | 26/0 | 26/0 | 51/51 | 100 | - | - | - | 0.94 | ⟨11.96, 6.31⟩ | ⟨6.15, 3.81⟩ |
| | stats | 121 | 121/0 | 121/0 | 150/150 | 91.15 | - | - | - | 0.85 | ⟨16.82, 9.44⟩ | ⟨8.12, 6⟩ |
| | textrank | 8 | 8/0 | 8/0 | 12/12 | 100 | - | - | - | 0.91 | ⟨13.75, 15.25⟩ | ⟨10.88, 11.88⟩ |
| **Total** | | 229 | 229/0 | 229/0 | 314/314 | 95.93 | - | - | - | 0.92 | ⟨20.21, 20.51⟩ | ⟨10.65, 8.84⟩ |
| AlphaTrans (Java→Python) | cli | 381 | 381/0 | 381/0 | 452/452 | 99.61 | 99.68 | 98.37 | 97.87 | 0.90 | ⟨12.41, 11.59⟩ | ⟨13.30, 12.63⟩ |
| | csv | 298 | 298/0 | 292/6 | 207/207 | 100 | 81.82 | 84.31 | 92.86 | 0.90 | ⟨11.84, 8.94⟩ | ⟨12.20, 10.46⟩ |
| | fileupload | 39 | 39/0 | 39/0 | 37/37 | 100 | 100 | 80 | 100 | 0.87 | ⟨6.74, 7.38⟩ | ⟨5.87, 6.36⟩ |
| | validator | 463 | 463/0 | 458/5 | 374/374 | 98.52 | 99.28 | 99.49 | 90.72 | 0.89 | ⟨17.69, 17.23⟩ | ⟨18.78, 18.30⟩ |
| **Total** | | 1181 | 1181/0 | 1170/11 | 1070/1070 | 99.53 | 95.20 | 90.54 | 95.36 | 0.89 | ⟨12.17, 11.29⟩ | ⟨12.54, 11.94⟩ |
| Skel (Python→JavaScript) | bst | 11 | 11/0 | 11/0 | 74/74 | 100 | - | - | - | 0.91 | ⟨22.27, 22.64⟩ | ⟨5.73, 12⟩ |
| | colorsys | 2 | 2/0 | 2/0 | 39/39 | 100 | - | - | - | 0.93 | ⟨67, 65⟩ | ⟨45, 44⟩ |
| | heapq | 8 | 8/0 | 8/0 | 29/29 | 100 | - | - | - | 0.90 | ⟨13.12, 13.38⟩ | ⟨11.25, 10.25⟩ |
| | html | 7 | 7/0 | 7/0 | 19/19 | 100 | - | - | - | 0.92 | ⟨24.71, 22.71⟩ | ⟨20.29, 19.29⟩ |
| | mathgen | 5 | 5/0 | 5/0 | 163/163 | 100 | - | - | - | 0.93 | ⟨47, 51⟩ | ⟨49, 42⟩ |
| | rbt | 10 | 10/0 | 10/0 | 16/16 | 100 | - | - | - | 0.91 | ⟨17.90, 19.60⟩ | ⟨20.90, 17.10⟩ |
| | strsim | 19 | 19/0 | 19/0 | 150/150 | 100 | - | - | - | 0.93 | ⟨18.74, 17.21⟩ | ⟨22.84, 21.84⟩ |
| | toml | 12 | 12/0 | 12/0 | 17/17 | 100 | - | - | - | 0.90 | ⟨8.67, 8.75⟩ | ⟨6.08, 7⟩ |
| **Total** | | 74 | 74/0 | 74/0 | 507/507 | 100 | - | - | - | 0.92 | ⟨27.43, 27.54⟩ | ⟨22.64, 21.69⟩ |
| **Total** | | 1484 | 1484/0 | 1473/11 | 1891/1891 | 98.54 | 95.20 | 90.54 | 95.36 | 0.91 | ⟨21.63, 21.58⟩ | ⟨16.40, 15.24⟩ |

*Validator* agent can delegate function-level equivalence checking when test execution alone is inconclusive or when a translated function remains uncovered. Concretely, MatchFixAgent can be wrapped as a Model Context Protocol (MCP) tool—consistent with the MCP-based tool surface already used by ReCodeAgent (§7.3.1)—and presented to the *Validator* agent alongside existing LSP and project-analysis tools. The agent would supply a source–translation function pair and receive structured verdicts, generated tests, and repair suggestions in return, enabling a two-stage workflow in which ReCodeAgent produces an initial repository translation and MatchFixAgent certifies or repairs individual functions that remain suspect. Such a composition could raise overall success rates beyond what either system achieves in isolation, although realizing it requires addressing orchestration overhead, reconciling function-granular repairs with repository-level translation state, and avoiding redundant test generation across the two pipelines.

### 7.4.3 RQ2: Test Translation

Table 22 shows the results of ReCodeAgent's test translation capabilities, comparing translated tests against their original source PL counterparts. This evaluation covers

1,484 tests across three PL pairs, excluding the CRUST benchmark, as it does not require test translation. RECODEAGENT successfully translates all 1,484 source tests to their target PLs, achieving 100% translation rate across all benchmarks, demonstrating the ability of RECODEAGENT's Validator Agent to ensure all tests are properly translated with no empty test logic. For translated tests to be semantically equivalent, they should preserve the same number of assertions as the original tests. Across all benchmarks, $\frac{1{,}473}{1{,}484}$ tests (99.3%) have matching assertion counts between source and translated PLs. Only 11 tests exhibit non-matching counts, all in ALPHATRANS projects. These cases occur when source tests contain a significantly large number of assert statements (e.g., 50+) due to hallucination.

For `assertEqual`-style assertions, we evaluate whether translated tests have the same expected values as original tests. Specifically, we check if expected outputs are similar for four types, `string, int, float, bool`. RECODEAGENT achieves 100% matching on `assertEqual` outputs, with all 1,891 assertions producing equivalent expected values in target PL. This metric is critical because `assertEqual` assertions directly validate functional correctness—if a translated test expects a different output value, it would fail even on a correct translation. We also evaluate whether RECODEAGENT preserves semantic types of assertions during translation. For instance, we check if `assertEqual(a, b)` in Java is translated to `assertEqual(a, b)` or `assertTrue(a)` when `b` is a boolean in Python. Multi-column *Assertion Type Match* shows the results of this evaluation. For `assertEqual` assertions, RECODEAGENT achieves 98.54% match rate across all benchmarks. Specifically, OXIDIZER achieves 95.93% and SKEL achieves 100%. For ALPHATRANS, which tests a broader variety of assertion types due to JUnit's rich assertion library, RECODEAGENT achieves: 99.53% for `assertEqual`, 95.20% for `assertTrue`, 90.54% for `assertFalse`, and 95.36% for other assertions including `assertNull` and `assertThrows`. The lower match rate for `assertFalse` (90.54%) is because of the translation of certain Java assertion idioms to semantically equivalent but syntactically different Python expressions. For instance, `assertFalse(list.isEmpty())` in Java is translated to `assert len(list) > 0` in Python, which tests the same condition but uses a different assertion pattern.

Moreover, we evaluate semantic similarity between source and translated tests using cosine similarity computed over code embeddings generated by `Qwen/Qwen3-Embedding-0.6B` [262]. This metric captures the degree to which translated tests preserve structural and logical patterns, independent of surface-level syntactic differences between languages. Across all benchmarks, ReCodeAgent achieves an average cosine similarity of 0.91, indicating high semantic preservation. We also compare structural characteristics using lines of code (LoC) and method invocation counts. On average, source tests contain 21.63 LoC while translated tests contain 21.58 LoC, a difference of less than 1%. This alignment indicates that ReCodeAgent produces translations of comparable complexity without code bloat or oversimplification. For method invocations, source tests average 16.40 invocations while translated tests average 15.24, a reduction of 7%. This reduction is attributed to idiomatic differences between testing frameworks.

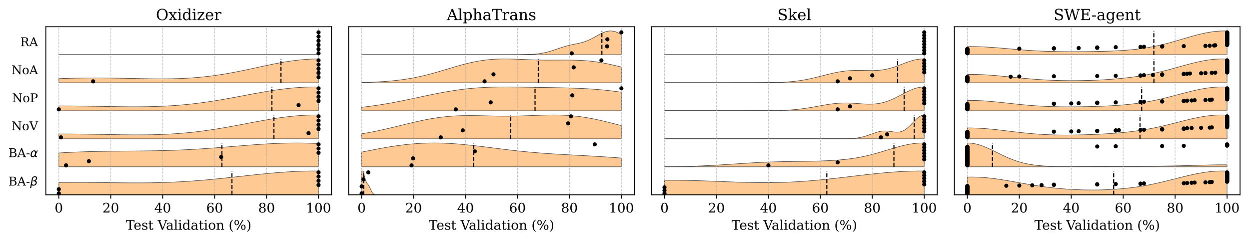
Taken together, these results show that ReCodeAgent preserves test structure and assertion-level behavior at high rates on the benchmarks studied. However, the metrics above primarily compare *surface* artifacts—assertion counts, expected values, assertion types, embedding similarity, and coarse structural statistics—rather than the *intent* that a test encodes. This limitation is most visible on tests with elaborate setup, staged inputs, multi-step execution sequences, or rich assertion logic. For example, the 90.54% assertion type match rate for `assertFalse` reflects cases where a translated test checks the same condition through a syntactically different but semantically equivalent pattern (e.g., `assertFalse(list.isEmpty())` versus `assert len(list) > 0`), which our current metrics treat as a mismatch even though the underlying intent may be preserved. Conversely, a translated test can match on assertion count, expected values, and execution outcome while silently weakening intent—for instance, by collapsing a sequence of boundary-case checks into a single aggregate assertion, or by replacing explicit state setup with framework-specific fixtures that exercise a narrower slice of behavior. A test that passes on a correct translation therefore does not, by itself, establish that the translated test preserves the source test's intent.

Characterizing and validating test intent more explicitly is a promising direction for future research. Building on the patterns surfaced in this evaluation, future work could develop intermediate representations that decompose a test into its setup preconditions, input sequences, observable interactions, and assertion obligations, and then define intent-preservation checks that operate on those representations rather than on raw syntax or pass/fail outcomes alone. Such techniques could complement the execution-based validation used throughout this dissertation by detecting when translated tests are *behaviorally* adequate yet *intent-deficient*—a failure mode that is especially likely as test complexity grows beyond the relatively simple cases that dominate the Oxidizer and Skel benchmarks.

### 7.4.4 RQ3: Ablation Study

Figure 28 presents the results of our ablation studies, evaluating the contribution of each component in ReCodeAgent. We compare ReCodeAgent (RA) against five ablated configurations: No Analyzer (NoA), No Planning (NoP), No Validator (NoV), Base Agent with Prompt Condensation (BA-$\alpha$), and Base Agent with Prompt Concatenation (BA-$\beta$). The top portion of Figure 28 shows test validation percentages and their distribution across all four benchmarks, while the bottom portion presents trajectory analysis using process-centric metrics, demonstrating agent behavior patterns.

**Impact of Individual Agents.** Removing the analyzer agent (NoA) results in decreased test validation performance ($\downarrow$22.7%) across all benchmarks. Without foundational analysis of the source project structure and third-party library




**Oxidizer**

| | NC | TEC | SEC | LC | ALL |
|---|---|---|---|---|---|
| RA | 88.3 | 107.6 | 8.8 | 316.6 | 11.9 |
| NoA | 170.3 | 210.8 | 23.4 | 696.3 | 21.0 |
| NoP | 143.6 | 179.9 | 22.2 | 181.6 | 18.3 |
| NoV | 255.3 | 334.3 | 28.0 | 1418.2 | 33.3 |
| BA-α | 95.2 | 95.0 | 0.0 | 0.8 | 2.4 |
| BA-β | 173.7 | 185.3 | 17.5 | 129.0 | 38.5 |

**AlphaTrans**

| | NC | TEC | SEC | LC | ALL |
|---|---|---|---|---|---|
| RA | 286.6 | 316.9 | 43.7 | 237.6 | 27.2 |
| NoA | 293.6 | 330.8 | 68.8 | 605.2 | 24.4 |
| NoP | 757.5 | 858.5 | 142.3 | 979.9 | 64.6 |
| NoV | 519.8 | 624.8 | 89.5 | 802.8 | 49.5 |
| BA-α | 331.0 | 345.8 | 41.5 | 45.5 | 46.8 |
| BA-β | 121.2 | 134.5 | 15.8 | 12.8 | 34.5 |

**Skel**

| | NC | TEC | SEC | LC | ALL |
|---|---|---|---|---|---|
| RA | 139.4 | 172.6 | 15.5 | 924.9 | 8.6 |
| NoA | 165.5 | 205.9 | 13.6 | 252.1 | 12.3 |
| NoP | 74.0 | 82.3 | 6.2 | 50.0 | 5.0 |
| NoV | 263.2 | 315.2 | 39.4 | 240.0 | 20.0 |
| BA-α | 51.3 | 50.6 | 0.0 | 0.3 | 1.2 |
| BA-β | 64.0 | 64.4 | 2.4 | 1.4 | 6.8 |

**SWE-agent**

| | NC | TEC | SEC | LC | ALL |
|---|---|---|---|---|---|
| RA | 89.6 | 102.0 | 15.0 | 164.2 | 11.7 |
| NoA | 83.2 | 94.6 | 11.4 | 196.1 | 10.4 |
| NoP | 53.4 | 56.8 | 5.7 | 75.1 | 6.6 |
| NoV | 116.8 | 137.5 | 20.2 | 426.1 | 15.1 |
| BA-α | 93.6 | 97.5 | 5.7 | 69.7 | 11.9 |
| BA-β | 88.2 | 92.4 | 5.0 | 9.4 | 14.5 |

Figure 28: Impact of different agents on translation effectiveness and agent trajectories. RA: ReCodeAgent, NoA: No Analyzer, NoP: No Planning, NoV: No Validator, BA-$\alpha$: Base Agent with Prompt Condensation, BA-$\beta$: Base Agent with Prompt Concatenation, NC: Node Count, TEC: Temporal Edge Count, SEC: Structural Edge Count, LC: Loop Count, ALL: Average Loop Length.

dependencies, the translator and validator agents must repeatedly explore the codebase, exhausting the context window and leading to inefficient interactions. The removal of the planning agent (NoP) demonstrates even more significant performance degradation (↓25.3%), as the translator agent lacks structured guidance for translation ordering, often resulting in repeated attempts. Finally, removing the validator agent (NoV) leads to the highest decrease in test validation (↓30.3%), as translation errors accumulate without dedicated test execution and diagnostic feedback.

**Comparison with Base Agents.** The base agent configurations represent ReCodeAgent without specialized agents. BA-$\alpha$ condenses the entire translation task into a single compact prompt, while BA-$\beta$ concatenates all ReCodeAgent prompts into a large prompt. Both perform significantly worse than ReCodeAgent across all benchmarks, by as much as 61.2% for BA-$\alpha$ and 62.4% for BA-$\beta$. These results demonstrate that simply providing an LLM with all available information does not yield effective translation—the structured decomposition into analysis, planning, translation, and validation phases is essential.

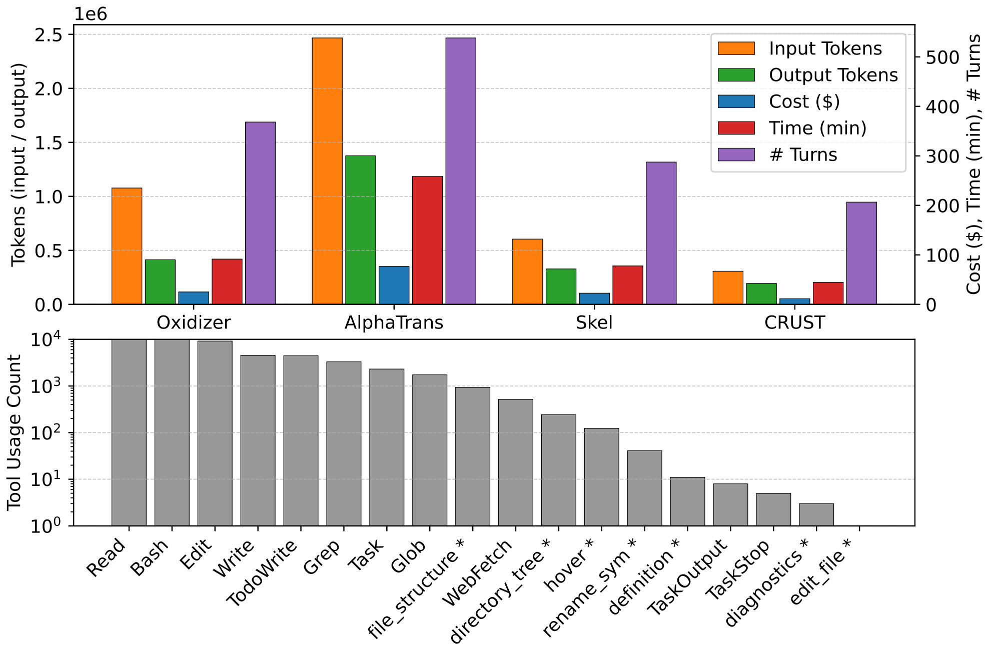



Figure 29: Cost and Tool Usage Analysis of ReCodeAgent.

**Trajectory Analysis.** To perform a process-centric analysis of agent trajectories, we use Graphectory [39]. Our objective is to show that test validation degradation alone is insufficient for evaluating ablations. The heatmaps in Figure 28 reveal distinct patterns in agent behavior. ReCodeAgent exhibits the most compact trajectories with the lowest average node and temporal edge counts, while achieving high test validation. The ablated configurations show progressively larger trajectory footprints as components are removed, up to 29% and 27% more node count (NC) and temporal edge count (TEC). These process-centric metrics consistently show that ReCodeAgent's multi-agent architecture provides structured guidance that prevents unnecessary exploration and repeated work.

In summary, all three specialized agents contribute substantially to ReCodeAgent's effectiveness, and removing any single component leads to significant degradation in both translation quality and efficiency.

### 7.4.5 RQ4: Cost and Tool Usage Analysis

Figure 29 presents RECODEAGENT's computational costs per project and tool utilization patterns across all four benchmarks.

**Cost.** Project costs and token usage scale with complexity, ranging from ALPHATRANS (2.5M input/1.4M output tokens at $76) and OXIDIZER (1.1M input/0.4M output at $25) to the more economical SKEL (0.6M input/0.3M output at $20) and CRUST (0.3M input/0.2M output at $11). Execution time scales linearly with project size: ALPHATRANS averages 258 minutes per project, OXIDIZER 92 minutes, SKEL 78 minutes, and CRUST 45 minutes due to smaller individual project sizes. This predictable scaling enables users to estimate costs for new projects based on codebase size.

**Tool Usage Distribution.** The bottom panel shows tool invocation patterns capped at $10K$. Core tools—`Read`, `Bash`, and `Edit`—are each invoked approximately 15,000 times on average for examining code, executing commands, and modifying files. `Write` and `Grep` support file creation and search operations. RECODEAGENT's LSP tools demonstrate targeted utilization: `file_structure` (∼1,000 invocations), `hover` for type information (∼150), and semantic tools like `definition` (∼40) enable reliable code navigation.

In summary, RECODEAGENT is economically viable with costs scaling linearly with project complexity, positioning it as a practical alternative to heavily engineered neuro-symbolic approaches that require 3,843–19,052 LoC of PL-specific implementation.

## 7.5 Related Work

**Code Translation.** There are two main approaches for translating code from one PL to another: traditional rule-based transpiler techniques and LLMs. Transpiler tools like C2Rust [18], CxGo [19], Sharpen [20], and Java2CSharp [21] translate code from C to Rust, C to Go, and Java to C#, respectively. A series of statistical machine translation techniques [24], [25], [26], [27] focus on translating Java to C#. Deep learning approaches have also been applied for code translation [29], [30]. Recent advancements have focused on using LLMs for code translation [22], [56], [57], [58], [59], [60], which have demonstrated strong performance on synthetic benchmarks but limited effectiveness on real-world software projects. Furthermore, repository-level code translation has been studied for various PL pairs. AlphaTrans [45] translates Java to Python using open-source LLMs and GraalVM [91] for isolated validation; Syzygy [75] targets C to Rust using GPT-4; Skel [77] translates Python to JavaScript; Oxidizer [76] employs type-driven techniques and language feature mapping to convert Go to Rust. Some approaches have combined transpiler outputs with LLM-based translation [85], but their success is often limited by the availability and reliability of the underlying transpilers. Nitin et al. [84] capture natural language specifications from source code to inform translation, while Yang et al. [86] utilize test cases to support the process.

**LLM Agents.** The rise of agent-based frameworks [246], [247] has produced significant research and industrial interest in applying these architectures to a variety of software engineering challenges [97], [98], [99]. SWE-agent [100] introduces a specialized agent-computer interface (ACI), enabling agents to interact with code repositories through file reading, editing, and execution of bash commands. AutoCodeRover [101] equips LLM agents with dedicated code-search APIs, supporting iterative retrieval and localization of code fragments related to software bugs. Building on this, SpecRover [102] extends AutoCodeRover by focusing on specification inference, generating function summaries, and offering targeted feedback at key points in the agent's workflow. Agentless [103] demonstrates that even simple LLM agents can address real-world bugs without extensive toolchains or complex modeling of environment behavior. In addition to these leading frameworks, a variety of other agent-based approaches are available in both open-source [105], [106], [135] and commercial solutions [104].

## 7.6 Threats to Validity

Similar to prior techniques, RECODEAGENT comes with some limitations and threats to validity. In this section, we discuss how we mitigated various threats.

**Internal Validity.** A major internal threat is that we run each experiment once. As LLMs are non-deterministic, repeated runs may yield different individual test outcomes. However, given the scale of our evaluation (118 projects), aggregate metrics are unlikely to change significantly.

**External Validity.** The primary external threat concerns generalizability. To mitigate this, RECODEAGENT is designed to be PL-agnostic, requiring minimal engineering effort to extend to new PL pairs. Our initial implementation supports six PLs across four PL pairs. A secondary threat is data contamination, as our benchmark programs were likely included in Claude's pre-training data, potentially inflating apparent performance.

**Construct Validity.** To minimize construct validity threats, RECODEAGENT is built upon well-vetted, widely adopted tools, including Tree-sitter and Claude Code.

## 7.7 Conclusion

In this work, we introduced RECODEAGENT, a *language-agnostic repository-level* code translation and validation framework that integrates four specialized LLM agents to achieve high-quality translations validated by both developer-written and agent-generated tests. RECODEAGENT combines the power of LLMs with reliable static analysis tools to translate projects across four different language pairs and six distinct PLs. To the best of our knowledge, RECODEAGENT is the first approach that can effectively translate and validate code at the repository level across multiple PLs.

# Chapter 8 Conclusion and Future Work

The lifespan of software routinely exceeds that of the PL in which it was first written, and the cost of carrying that mismatch forward—legacy applications written in PLs that the workforce no longer knows, support ecosystems that no longer receive security updates, library stacks that no longer fit a cloud-native deployment—is paid in human hours, security incidents, and missed business opportunities. Automated code translation has therefore been a persistent goal of PLs and software engineering research for more than two decades. The rise of LLMs reset the outlook for that goal: for the first time, a single model could translate between PLs that it had never been explicitly paired on, handle libraries whose APIs were only implicitly present in its training data, and produce target PL code that was idiomatic at the line level. Yet, as this dissertation has shown, off-the-shelf LLM translation collapses on the codebases that matter most—real-world repositories with cross-file dependencies, third-party libraries, and developer-written test suites—and the path back to practicality runs through a careful, staged composition of LLMs with classical and approximate program analyses.

This dissertation has developed and evaluated that staged composition end-to-end. The empirical investigation of Chapter 4 characterizes *what goes wrong* when LLMs translate real-world code, deriving a 15-category bug taxonomy from 630 person-hours of manual labeling of 1,748 buggy translations and quantifying the limited effectiveness of iterative prompt engineering as a mitigation. AlphaTrans (Chapter 5) translates the bug taxonomy into action for a single PL pair through a neuro-symbolic compositional pipeline that decomposes a repository into syntactic fragments, translates them in reverse call order, and validates each fragment in isolation through GraalVM-based language interoperability—establishing the first technique able to translate *and* validate complete real-world repositories including their tests. MatchFixAgent (Chapter 6) loosens AlphaTrans's PL-specific symbolic infrastructure into approximate, language-agnostic semantic analyses behind a multi-agent validation-and-repair framework, repairing translation bugs across six PL pairs at roughly 280 lines of PL-specific code per pair—two orders of magnitude below comparable systems. Finally, ReCodeAgent (Chapter 7) extends the same PL-agnostic principles to the translation stage itself through a four agent (*Analyzer*, *Planning*, *Translator*, *Validator*) pipeline grounded in Tree-sitter and Language Server Protocols, achieving 99.4% compilation and 86.5% test pass rates across four PL pairs and 118 real-world projects at an average cost of $15.30 and 57 minutes per project.

In the remainder of this chapter, I conclude this dissertation by enumerating its contributions to the software engineering research community (§8.1) and by outlining concrete avenues for future work (§8.2).

## 8.1 Conclusion

This dissertation makes the following contributions to the software engineering research community, organized into four categories: theoretical and methodological frameworks, empirical findings, tools, and publicly released datasets and artifacts.

- **Theoretical and Methodological Frameworks:**

  1. *The first* 15*-category,* 5*-group bug taxonomy for LLM-based code translation.* Chapter 4 introduces a methodologically rigorous taxonomy of LLM translation failures, derived through an open-coding process applied by eight independent labelers to 1,748 buggy GPT-4 translations and stabilized across seven LLMs, five PLs, and four benchmarks. The taxonomy is the first published vocabulary of failure modes for LLM-based code translation and has been adopted as a debugging lens by every subsequent chapter of this dissertation.

  2. *A neuro-symbolic compositional framework for repository-level translation and validation.* Chapter 5 introduces a framework that decomposes a source

project into syntactic fragments via static analysis, translates them in reverse call-graph order using an LLM, and validates each translated fragment in isolation through cross-language interoperability while the rest of the project remains in the source PL. The framework is the first to treat the test suite as a translation target on equal footing with application code and the first to use language interoperability for in-isolation fragment validation.

3. *A language-agnostic, multi-agent framework for repository-level validation and repair.* Chapter 6 introduces a framework in which approximate, PL-agnostic semantic analyses—control-flow, data-flow, library API, exception, and specification analyses—are placed behind specialized verdict and repair agents whose role separation is itself a principled response to known reward hacking failure modes of single-agent pipelines. The framework reduces the per PL-pair engineering cost of validation-and-repair from thousands of lines of PL-specific code to roughly 280 lines, while improving repair accuracy over the strongest re-prompting baseline by 32.1%.

4. *A multi-agent, end-to-end framework for language-agnostic repository-level translation.* Chapter 7 introduces a four agent framework—*Analyzer*, *Planning*, *Translator*, and *Validator*—in which every agent is grounded in PL-agnostic semantic tooling (Tree-sitter and Language Server Protocols) and in which the Validator inherits the validation-and-repair design of Chapter 6. The framework is the first to combine language-agnosticism with semantic grounding for repository-level translation *and* validation, and the first to expose the agentic design space along the process-centric metrics of Liu et al. [39].

- **Empirical Findings:**

1. *The first systematic evidence that LLM-based translation collapses on*

*real-world projects.* On two open-source command-line projects (Apache Commons CLI and Python Click), only GPT-4 achieved a non-zero functional correctness rate (8.1%); every other LLM scored 0%. Iterative prompt engineering improved success rates by an average of only 5.5% across studied LLMs and 12% for GPT-4, establishing that mitigation by prompting alone is insufficient.

2. *The first end-to-end repository-level translation results.* On ten real-world Java projects totaling 17,874 fragments, AlphaTrans achieves 96.40% syntactic correctness and 25.14% functional equivalence on 4,654 executable method fragments—rising to 99.2% and 27.95% respectively when GPT-4o is used—at an average cost of $14.39 per project. A human subject study further establishes that partial AlphaTrans translations can be brought to passing test suites in 20.1 hours of human effort on average, providing the first pragmatic dataset for downstream research on repository-level translation repair.

3. *The first cross-PL-pair validation-and-repair results at scale.* On 2,219 source–translation function pairs drawn from 24 real-world projects totaling more than 900,000 lines of code across six PL pairs, MatchFixAgent produces a verdict on 99.2% of pairs (versus 71.6% for prior PL-specific techniques) and repairs 50.6% of inequivalent translations. An ablation study demonstrates that removing either the semantic analyzer or the in-the-loop test generator component reduces verdict accuracy by up to 42.3% while *increasing* token usage, establishing both components as necessary.

4. *The first end-to-end agentic, language-agnostic translation results.* On 4,583 translation units drawn from 118 real-world projects totaling more than 230,000 lines of code across four PL pairs (C–Rust, Go–Rust, Java–Python, Python–JavaScript), ReCodeAgent achieves 99.4% compilation and 86.5%

test pass rates—2.5 and 60.8 percentage points above the strongest prior baselines, respectively—at an average cost of $15.30 and 57 minutes per project. When translating tests, RECODEAGENT achieves 99.3% assertion equivalence, 0.91 cosine similarity, and 94.9% assertion type match between source and target tests. Removing the Analyzer, Planning, or Validator agent reduces the test pass rate by 22.7%, 25.3%, or 30.3%, respectively, while increasing trajectory complexity by 28%.

- **Tools:** To enable other researchers and practitioners to reproduce, extend, and apply the techniques of this dissertation, all tools are released as open-source artifacts:

  1. ALPHATRANS, a neuro-symbolic Java-to-Python compositional translator and validator [41].
  2. MATCHFIXAGENT, a language-agnostic, multi-agent translation validator and repair framework [42].
  3. RECODEAGENT, a four agent, language-agnostic, end-to-end translation-and-validation pipeline [43].

- **Datasets and Artifacts:** To support follow-up empirical research and benchmark construction, the following datasets and artifacts are released alongside the tools above:

  1. The bug taxonomy dataset of 1,748 manually labeled GPT-4 buggy translations across five PLs, organized along the 15-category, 5-group taxonomy of Chapter 4 [40].
  2. The ALPHATRANS benchmark of ten real-world Java projects with their human repaired Python ground-truth counterparts, together with 17,874 syntactic fragments and 4,654 executable method fragments [41].

3. The MatchFixAgent cross-PL validation benchmark: 2,219 source–translation function pairs from 24 real-world projects spanning six PL pairs and more than 900,000 lines of code [42].
4. The ReCodeAgent trajectory dataset: complete agent trajectories, tool execution traces, and per-PL-pair compilation, test pass, and assertion fidelity outcomes for 4,583 translation units across 118 real-world projects and four PL pairs [43].

Taken together, these contributions establish that LLMs, when composed with the right program analyses—first taxonomically understood, then symbolically scaffolded for a single PL pair, and finally generalized through approximate, PL-agnostic tooling beneath a multi-agent architecture—can translate and validate complete real-world repositories across multiple source–target PL pairs at engineering and operational costs that are practical for both academic research and industrial software modernization efforts. The four chapters of this dissertation provide the techniques, the empirical evidence, and the publicly released artifacts that, jointly, support this thesis.

### 8.2 Future Work

By leveraging the techniques, empirical findings, and infrastructure developed in this dissertation, I am considering several directions for future work, each motivated by a limitation surfaced in one or more of the four research chapters.

- **Library API translation as a first-class research problem.** The bug taxonomy of Chapter 4 and the conclusion of Chapter 5 both identify library API mismatch as one of the most persistent failure modes of repository-level translation: an idiomatic counterpart to a source PL library may not exist in the target PL, may exist but expose subtly different semantics (e.g., default exception behavior, threading model, locale handling), or may exist only as a partial subset of the source PL functionality. A promising direction is to treat library API translation as a first-class research problem in its own right: given a source PL library $\ell_s$ used by $P_s$, automatically synthesize a target PL implementation of $\ell_s$'s relevant interface, validated by differential testing against $\ell_s$ on inputs drawn from the source PL test suite. The validation primitives of MatchFixAgent and the trajectory infrastructure of ReCodeAgent provide natural starting points for building such a system.

- **PL-agnostic static analysis as a first-class capability.** AlphaTrans demonstrates that heavy, PL-specific static analysis (CodeQL queries, custom AST passes, bespoke dependency extraction) suffices to translate real-world Java repositories, but its conclusion explicitly identifies the PL-specificity of that infrastructure as the principal obstacle to wider applicability. MatchFixAgent and ReCodeAgent partially address this through approximate, PL-agnostic semantic analyses (Tree-sitter and Language Server Protocols), but the analysis depth they currently expose is limited to symbol resolution, control-flow, and shallow data-flow. A natural future direction is to elevate PL-agnostic static analysis to a first-class capability of the agentic pipeline, by extending the Model Context Protocol with deeper analyses—inter-procedural data-flow, points-to analysis, and lightweight type inference—that have historically been considered too PL-specific to expose through a generic protocol.

- **Hybrid formal-and-agentic equivalence checking.** The validation strategies developed in Chapter 6 and Chapter 7 are sound on the inputs the test suite covers but, like all test execution-based validation, do not establish equivalence over the unbounded input space. Formal methods-based validation [84], [85] establishes stronger guarantees but at the cost of scalability. A promising hybrid direction is to combine the agentic translation pipeline of ReCodeAgent with formal equivalence checking on a targeted subset of high-stakes functions (e.g., kernel routines, cryptographic primitives, safety-critical control logic), using the agentic pipeline to translate the bulk of the repository and the formal checker to certify the small slice where formal guarantees matter most.

- **Translation validation beyond functional correctness.** Every validation strategy in this dissertation—GraalVM-based in-isolation validation in AlphaTrans, semantic analysis and test generation in MatchFixAgent, and the *Validator* agent of ReCodeAgent—targets *functional* equivalence with respect to developer-written or agent-generated tests. Preserving externally observable behavior is necessarily the primary objective of repository-level translation, but practical adoption also depends on whether translated code retains important non-functional properties such as throughput and latency, memory footprint, security posture (e.g., absence of newly introduced `unsafe` blocks or injection vulnerabilities), idiomaticity and maintainability, and compatibility with downstream build and deployment tooling. A promising future direction is to extend the validation-and-repair loop with complementary analyses and benchmarks that evaluate these properties alongside functional correctness—for example, by pairing differential performance testing on representative workloads, static security analyzers and type-system checks in the target PL, and maintainability metrics such as cyclomatic complexity and adherence to target-language style guides—and to investigate optimization passes that trade marginal functional risk for gains in non-functional quality when translation artifacts are deployed at industrial scale.

- **Industrial-scale modernization studies.** Every benchmark in this dissertation, including the 230,000+ lines of code spanned by ReCodeAgent, comes from open-source projects with developer-written test suites. Industrial codebases differ from open-source projects in three ways that this dissertation has not yet measured: they are typically an order of magnitude larger; they often contain undocumented internal APIs, build configurations, and deployment scripts; and their tests are frequently inadequate or non-public. A concrete future direction is to deploy ReCodeAgent on industrial codebases through partnerships with modernization teams, study the additional engineering required to handle these differences, and extend the bug taxonomy of Chapter 4 with the new failure modes that surface only at industrial scale.

- **Process-centric benchmarking for agentic software engineering systems.** ReCodeAgent reports its results not only along outcome metrics (compilation, test pass, assertion fidelity) but also along the process-centric metrics of Liu et al. [39]—node count, tool execution count, and trajectory entropy. The broader agentic SE community, including the SWE-bench [97], AutoCodeRover [101], OpenHands [104], and CRUST-Bench [109] ecosystems, has not yet adopted comparable reporting conventions. A promising community-level future direction is to standardize process-centric reporting across agentic SE benchmarks, possibly through shared trajectory schemas and a common cost-and-tool-use vocabulary, so that future agentic systems can be compared along the same dimensions that this dissertation applies to its own evaluation.

- **Translation provenance and human-in-the-loop debugging.** The agentic pipelines of MatchFixAgent and ReCodeAgent already produce structured trajectories, but those trajectories are not yet linked back to the resulting code in a way that lets a human debugger ask, "which agent decision, which tool invocation, and which prompt context produced *this* bug?" A natural future direction is to add translation provenance tracking to the agentic pipeline, so that each line of $F_t$ carries a back-reference to the agent step that produced it, and to build human-in-the-loop debugging interfaces around that provenance. The artifacts released alongside this dissertation already provide the trajectory data such interfaces would consume.

- **Beyond translation: agentic refactoring, modernization, and API evolution.** Translation is one instance of a broader class of *program-rewriting under constraints*: the input is a working program, the output is a different working program, and the rewrite must preserve some equivalence relation that may be looser than functional equivalence (e.g., preserving externally observable behavior while modernizing internal data structures, or preserving I/O behavior while migrating to a different cloud-native architecture). The four agent (*Analyzer*, *Planning*, *Translator*, *Validator*) architecture of ReCodeAgent is a natural starting point for these adjacent problems, with the *Translator* role generalized to a *Rewriter* that consumes a target-side specification or refactoring intent, and the *Validator* role generalized to a behavior-preservation checker.

## References


[1] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage, "How do professionals perceive legacy systems and software modernization?" In *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 36–47, ISBN: 9781450327565. DOI: 10.1145/2568225.2568318 [Online]. Available: https://doi.org/10.1145/2568225.2568318

[2] S. Jain and I. Chana, "Modernization of legacy systems: A generalised roadmap," in *Proceedings of the Sixth International Conference on Computer and Communication Technology 2015*, ser. ICCCT '15, Allahabad, India: Association for Computing Machinery, 2015, pp. 62–67, ISBN: 9781450335522. DOI: 10.1145/2818567.2818579 [Online]. Available: https://doi.org/10.1145/2818567.2818579

[3] P. Jamshidi, A. Ahmad, and C. Pahl, "Cloud migration research: A systematic review," *IEEE Transactions on Cloud Computing*, vol. 1, no. 2, pp. 142–157, 2013. DOI: 10.1109/TCC.2013.10

[4] J. Thönes, "Microservices," *IEEE software*, vol. 32, no. 1, pp. 116–116, 2015.

[5] W. Nisar, "Modernization framework to enhance the security of legacy information systems," *Intelligent Automation & Soft Computing*, 2022.

[6] S. G. Haugeland, P. H. Nguyen, H. Song, and F. Chauvel, "Migrating monoliths to microservices-based customizable multi-tenant cloud-native apps," in *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, IEEE, 2021, pp. 170–177.

[7] J. Kazanavičius and D. Mažeika, "Migrating legacy software to microservices architecture," in *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, IEEE, 2019, pp. 1–5.

[8] H. Pei Breivold, “Towards factories of the future: Migration of industrial legacy automation systems in the cloud computing and internet-of-things context,” *Enterprise Information Systems*, vol. 14, no. 4, pp. 542–562, 2020.

[9] A. Bergmayr et al., “Migrating legacy software to the cloud with artist,” in *2013 17th European Conference on Software Maintenance and Reengineering*, IEEE, 2013, pp. 465–468.

[10] W. Zhang, A. J. Berre, D. Roman, and H. A. Huru, “Migrating legacy applications to the service cloud,” in *Proceedings of the 14th Conference Companion on Object Oriented Programming Systems Languages and Applications*, 2009, pp. 59–68.

[11] M. F. Gholami, F. Daneshgar, G. Beydoun, and F. Rabhi, “Challenges in migrating legacy software systems to the cloud—an empirical study,” *Information Systems*, vol. 67, pp. 100–113, 2017.

[12] A. K. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, and D. Banerjee, “Mono2micro: A practical and effective tool for decomposing monolithic java applications to microservices,” in *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 1214–1224.

[13] V. Nitin, S. Asthana, B. Ray, and R. Krishna, “Cargo: Ai-guided dependency analysis for migrating monolithic applications to microservices architecture,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[14] R. Krishna, A. Kalia, S. Sinha, R. Tzoref-Brill, J. Rofrano, and J. Xiao, “Transforming monolithic applications to microservices with mono2micro,” in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, 2021, pp. 3–3.

[15] R. Pérez-Castillo, M. A. Serrano, and M. Piattini, “Software modernization to embrace quantum technology,” *Advances in Engineering Software*, vol. 151, p. 102 933, 2021.

[16] R. Settu and P. Raj, “Cloud application modernization and migration methodology,” *Cloud Computing: Methods and Practical Approaches*, pp. 243–271, 2013.

[17] R. R. Echeverria, F. Macias, V. M. Pavon, J. M. Conejero, and F. S. Figueroa, “Legacy web application modernization by generating a rest service layer,” *IEEE Latin America Transactions*, vol. 13, no. 7, pp. 2379–2383, 2015.

[18] Immunant, *C2rust transpiler*, 2024. [Online]. Available: `https://github.com/immunant/c2rust`

[19] G. Transpile, *C to go translator*, 2024. [Online]. Available: `https://github.com/gotranspile/cxgo`

[20] M. Project, *Sharpen - automated java->c# coversion*, 2026. [Online]. Available: `https://github.com/mono/sharpen`

[21] P. Irwin, *Java to csharp converter*, 2026. [Online]. Available: `https://github.com/paulirwin/JavaToCSharp`

[22] R. Pan et al., “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*, ser. ICSE ’24, Lisbon, Portugal: Association for Computing Machinery, 2024, ISBN: 9798400702174. DOI: `10.1145/3597503.3639226` [Online]. Available: `https://doi.org/10.1145/3597503.3639226`

[23] A. Bastidas Fuertes, M. Pérez, and J. Meza Hormaza, “Transpilers: A systematic mapping review of their usage in research and industry,” *Applied Sciences*, vol. 13,

no. 6, 2023, ISSN: 2076-3417. DOI: 10.3390/app13063667 [Online]. Available: https://www.mdpi.com/2076-3417/13/6/3667

[24] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *FSE*, New York, NY, USA: ACM, 2013, pp. 651–654.

[25] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, Saint Petersburg, Russia: Association for Computing Machinery, 2013, pp. 651–654, ISBN: 9781450322379. DOI: 10.1145/2491411.2494584 [Online]. Available: https://doi.org/10.1145/2491411.2494584

[26] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-conquer approach for multi-phase statistical migration for source code," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15, Lincoln, Nebraska: IEEE Press, 2015, pp. 585–596, ISBN: 9781509000241. DOI: 10.1109/ASE.2015.74 [Online]. Available: https://doi.org/10.1109/ASE.2015.74

[27] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18, Montréal, Canada: Curran Associates Inc., 2018, pp. 2552–2562.

[28] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2014, pp. 173–184.

[29] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20, Vancouver, BC, Canada: Curran Associates Inc., 2020, ISBN: 9781713829546.

[30] B. Roziere, J. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample, "Leveraging automated unit tests for unsupervised code translation," in *International Conference on Learning Representations*, 2022. [Online]. Available: `https://openreview.net/forum?id=cmt-6KtR4c4`

[31] M.-A. Lachaux, B. Roziere, M. Szafraniec, and G. Lample, "Dobf: A deobfuscation pre-training objective for programming languages," *Advances in Neural Information Processing Systems*, vol. 34, pp. 14 967–14 979, 2021.

[32] OpenAI, *Gpt-4 technical report*, 2023. arXiv: `2303.08774`.

[33] T. A. Team, *Claude*, 2026. [Online]. Available: `https://www.anthropic.com/claude`

[34] R. Li et al., "Starcoder: May the source be with you!" *Transactions on Machine Learning Research*, 2023, Reproducibility Certification, ISSN: 2835-8856. [Online]. Available: `https://openreview.net/forum?id=KoFOg41haE`

[35] E. Nijkamp et al., "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations*, 2022.

[36] Q. Zheng et al., "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ser. KDD '23, Long Beach, CA, USA: Association for Computing Machinery, 2023, pp. 5673–5684, ISBN: 9798400701030. DOI: `10.1145/3580305.3599790` [Online]. Available: `https://doi.org/10.1145/3580305.3599790`

[37] D. Guo et al., “Deepseek-coder: When the large language model meets programming–the rise of code intelligence,” *arXiv preprint arXiv:2401.14196*, 2024.

[38] N. F. Liu et al., “Lost in the middle: How language models use long contexts,” *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024. DOI: 10.1162/tacl_a_00638 [Online]. Available: https://aclanthology.org/2024.tacl-1.9/

[39] S. Liu, Y. Chen, R. Krishna, S. Sinha, J. Ganhotra, and R. Jabbarvand, “Process-centric analysis of agentic software systems,” *Proc. ACM Program. Lang.*, vol. 10, no. OOPSLA1, Apr. 2026. DOI: 10.1145/3798271 [Online]. Available: https://doi.org/10.1145/3798271

[40] *Artifact website*, https://github.com/Intelligent-CAT-Lab/PLTranslationEmpirical, 2023.

[41] I. C. Lab, *Alphatrans artifact website*, https://github.com/Intelligent-CAT-Lab/AlphaTrans, 2025.

[42] MatchFixAgent, *Artifact website*, https://doi.org/10.5281/zenodo.17051107, 2025.

[43] ReCodeAgent, *Artifact website*, https://doi.org/10.5281/zenodo.19337799, 2026.

[44] A. R. Ibrahimzada, “Program decomposition and translation with static analysis,” in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE-Companion ’24, Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 453–455, ISBN: 9798400705021. DOI: 10.1145/3639478.3641226 [Online]. Available: https://doi.org/10.1145/3639478.3641226

[45] A. R. Ibrahimzada et al., "Alphatrans: A neuro-symbolic compositional approach for repository-level code translation and validation," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025. DOI: 10.1145/3729379 [Online]. Available: https://doi.org/10.1145/3729379

[46] A. R. Ibrahimzada, B. Paulsen, R. Jabbarvand, J. Dodds, and D. Kroening, "Matchfixagent: Language-agnostic autonomous repository-level code translation validation and repair," in *Forty-third International Conference on Machine Learning*, 2026. [Online]. Available: https://openreview.net/forum?id=MuyXpH3GL1

[47] A. R. Ibrahimzada, B. Paulsen, D. Kroening, and R. Jabbarvand, "Recodeagent: A multi-agent workflow for language-agnostic translation and validation of large-scale repositories," *arXiv preprint arXiv:2604.07341*, 2026.

[48] A. R. Ibrahimzada, Y. Chen, R. Rong, and R. Jabbarvand, "Challenging bug prediction and repair models with synthetic bugs," in *2025 IEEE International Conference on Source Code Analysis & Manipulation (SCAM)*, 2025, pp. 133–144. DOI: 10.1109/SCAM67354.2025.00021

[49] A. R. Ibrahimzada, Y. Varli, D. Tekinoglu, and R. Jabbarvand, "Perfect is the enemy of test oracle," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, Singapore, Singapore: Association for Computing Machinery, 2022, pp. 70–81, ISBN: 9781450394130. DOI: 10.1145/3540250.3549086 [Online]. Available: https://doi.org/10.1145/3540250.3549086

[50] K. Ke, A. R. Ibrahimzada, R. Pan, S. Sinha, and R. Jabbarvand, "Advancing automated in-isolation validation in repository-level code translation," *arXiv preprint arXiv:2511.21878*, 2025.

[51] L. B. Allal et al., “Santacoder: Don’t reach for the stars!” *arXiv preprint arXiv:2301.03988*, 2023.

[52] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, “A systematic evaluation of large language models of code,” in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[53] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, “CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, Eds., Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708. DOI: `10.18653/v1/2021.emnlp-main.685` [Online]. Available: `https://aclanthology.org/2021.emnlp-main.685/`

[54] T. L. Scao et al., “Bloom: A 176b-parameter open-access multilingual language model,” *arXiv preprint arXiv:2211.05100*, 2022.

[55] *Llama-2*, `https://ai.meta.com/research/publications/llama-2-open-foundation-and-fine-tuned-chat-models/`, 2023.

[56] S. Tipirneni, M. Zhu, and C. K. Reddy, “Structcoder: Structure-aware transformer for code generation,” *ACM Trans. Knowl. Discov. Data*, vol. 18, no. 3, Jan. 2024, ISSN: 1556-4681. DOI: `10.1145/3636430` [Online]. Available: `https://doi.org/10.1145/3636430`

[57] P. Di et al., “Codefuse-13b: A pretrained multi-lingual code large language model,” in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP ’24, Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 418–429, ISBN: 9798400705014. DOI: `10.1145/3639477.3639719` [Online]. Available: `https://doi.org/10.1145/3639477.3639719`

[58] W. Yan, Y. Tian, Y. Li, Q. Chen, and W. Wang, "CodeTransOcean: A comprehensive multilingual benchmark for code translation," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, H. Bouamor, J. Pino, and K. Bali, Eds., Singapore: Association for Computational Linguistics, Dec. 2023, pp. 5067–5089. DOI: `10.18653/v1/2023.findings-emnlp.337` [Online]. Available: `https://aclanthology.org/2023.findings-emnlp.337/`

[59] X. Yin, C. Ni, T. N. Nguyen, S. Wang, and X. Yang, "Rectifier: Code translation with corrector via llms," *arXiv preprint arXiv:2407.07472*, 2024.

[60] M. Jiao, T. Yu, X. Li, G. Qiu, X. Gu, and B. Shen, "On the evaluation of neural code translation: Taxonomy and benchmark," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1529–1541. DOI: `10.1109/ASE56229.2023.00114`

[61] P. Jana, P. Jha, H. Ju, G. Kishore, A. Mahajan, and V. Ganesh, "Cotran: An llm-based code translator using reinforcement learning with feedback from compiler and symbolic execution," in *ECAI 2024*, IOS Press, 2024, pp. 4011–4018.

[62] L. Gong, J. Wang, and A. Cheung, "Adelt: Transpilation between deep learning frameworks," in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, ser. IJCAI '24, Jeju, Korea, 2024, ISBN: 978-1-956792-04-1. DOI: `10.24963/ijcai.2024/694` [Online]. Available: `https://doi.org/10.24963/ijcai.2024/694`

[63] Q. Sun, N. Chen, J. Wang, M. Gao, and X. Li, "TransCoder: Towards unified transferable code representation learning inspired by human skills," in *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, N. Calzolari, M.-Y. Kan, V. Hoste, A. Lenci, S. Sakti, and N. Xue, Eds., Torino, Italia: ELRA

and ICCL, May 2024, pp. 16 713–16 726. [Online]. Available: https://aclanthology.org/2024.lrec-main.1453/

[64] M. Zhu, K. Suresh, and C. K. Reddy, "Multilingual code snippets training for program translation," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 10, pp. 11 783–11 790, Jun. 2022. DOI: 10.1609/aaai.v36i10.21434 [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/21434

[65] https://commons.apache.org/proper/commons-cli, 2023.

[66] *Click*, https://click.palletsprojects.com/en/8.1.x/, 2023.

[67] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "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*, ser. NIPS '23, New Orleans, LA, USA: Curran Associates Inc., 2023.

[68] C. S. Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971.

[69] C. S. Xia and L. Zhang, "Conversational automated program repair," *arXiv preprint arXiv:2301.13246*, 2023.

[70] H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, "Repair is nearly generation: Multilingual program repair with llms," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, 2023, pp. 5131–5140.

[71] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds., Online: Association for Computational

Linguistics, Nov. 2020, pp. 1536–1547. DOI: 10.18653/v1/2020.findings-emnlp.139 [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139/

[72] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1482–1494. DOI: 10.1109/ICSE48619.2023.00129

[73] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, Dec. 2023, ISSN: 1049-331X. DOI: 10.1145/3631974 [Online]. Available: https://doi.org/10.1145/3631974

[74] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 01, pp. 34–67, Jan. 2019, ISSN: 1939-3520. DOI: 10.1109/TSE.2017.2755013

[75] M. Shetty, N. Jain, A. Godbole, S. A. Seshia, and K. Sen, "Syzygy: Dual code-test c to (safe) rust translation using llms and dynamic analysis," *arXiv preprint arXiv:2412.14234*, 2024.

[76] H. Zhang, C. David, M. Wang, B. Paulsen, and D. Kroening, "Scalable, validated code translation of entire projects using large language models," *Proc. ACM Program. Lang.*, vol. 9, no. PLDI, Jun. 2025. DOI: 10.1145/3729315 [Online]. Available: https://doi.org/10.1145/3729315

[77] B. Wang, T. Li, R. Li, U. Mathur, and P. Saxena, "Program skeletons for automated program translation," *Proc. ACM Program. Lang.*, vol. 9, no. PLDI, Jun. 2025. DOI: 10.1145/3729287 [Online]. Available: https://doi.org/10.1145/3729287

[78] V. Nitin, R. Krishna, L. L. d. Valle, and B. Ray, "C2saferrust: Transforming c projects into safer rust with neurosymbolic techniques," *IEEE Transactions on Software Engineering*, vol. 52, no. 2, pp. 618–630, 2026. DOI: 10.1109/TSE.2025.3641486

[79] X. Cai et al., "Rustmap: Towards project-scale c-to-rust migration via program analysis and llm," in *International Conference on Engineering of Complex Computer Systems*, Springer, 2025, pp. 283–302.

[80] F. Luo et al., "Integrating rules and semantics for llm-based c-to-rust translation," in *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2025, pp. 685–696. DOI: 10.1109/ICSME64153.2025.00069

[81] C. Wang et al., "Evoc2rust: A skeleton-guided framework for project-level c-to-rust translation," *arXiv preprint arXiv:2508.04295*, 2025.

[82] T. Zhou, H. Lin, S. Jha, M. Christodorescu, K. Levchenko, and V. Chandrasekaran, "LLM-driven multi-step translation from c to rust using static analysis," in *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, 2025. [Online]. Available: https://openreview.net/forum?id=i5MwgkeGZN

[83] Z. Yuan et al., "Project-level c-to-rust translation via synergistic integration of knowledge graphs and large language models," *arXiv preprint arXiv:2510.10956*, 2025.

[84] V. Nitin, R. Krishna, and B. Ray, "Spectra: Enhancing the code translation ability of language models by generating multi-modal specifications," *arXiv preprint arXiv:2405.18574*, 2024.

[85] A. Z. Yang, Y. Takashima, B. Paulsen, J. Dodds, and D. Kroening, "Vert: Polyglot verified equivalent rust transpilation with large language models," in *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Seoul, Korea, Republic of: IEEE Press, 2025, pp. 1453–1463.

DOI: 10.1109/ASE63991.2025.00123 [Online]. Available:
https://doi.org/10.1109/ASE63991.2025.00123

[86] Z. Yang et al., "Exploring and unleashing the power of large language models in automated code translation," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. DOI: 10.1145/3660778 [Online]. Available: https://doi.org/10.1145/3660778

[87] S. Dehghan, T. Sun, T. Wu, Z. Li, and R. Jabbarvand, "Translating large-scale c repositories to idiomatic rust," *arXiv preprint arXiv:2511.20617*, 2025.

[88] G. Ou, M. Liu, Y. Chen, Y. Wang, X. Peng, and Z. Zheng, "Rustrepotrans: Repository-level context code translation benchmark targeting rust," in *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2025, pp. 610–622. DOI: 10.1109/ASE63991.2025.00057

[89] P. Xue et al., "Classeval-t: Evaluating large language models in class-level code translation," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, Jun. 2025. DOI: 10.1145/3728940 [Online]. Available: https://doi.org/10.1145/3728940

[90] Y. Wang et al., "Effireasontrans: Rl-optimized reasoning for code translation," *IEEE Transactions on Software Engineering*, pp. 1–12, 2026. DOI: 10.1109/TSE.2026.3693795

[91] Oracle, *Graalvm*, https://www.graalvm.org, 2026.

[92] M. S. Abid et al., "Gluetest: Testing code translation via language interoperability," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2024, pp. 612–617. DOI: 10.1109/ICSME58944.2024.00061

[93] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.

[94] H. F. Eniser et al., “Towards translating real-world code with llms: A study of translating to rust,” *arXiv preprint arXiv:2405.11514*, 2024.

[95] P. Godefroid, M. Y. Levin, D. A. Molnar, et al., “Automated whitebox fuzz testing,” in *NDSS*, vol. 8, 2008, pp. 151–166.

[96] S. Yao et al., “React: Synergizing reasoning and acting in language models,” in *The eleventh international conference on learning representations*, 2022.

[97] C. E. Jimenez et al., “SWE-bench: Can language models resolve real-world github issues?” In *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: `https://openreview.net/forum?id=VTF8yNQM66`

[98] J. Yang et al., “SWE-bench multimodal: Do AI systems generalize to visual software domains?” In *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: `https://openreview.net/forum?id=riTiq3i21b`

[99] N. Chowdhury et al., *Introducing SWE-bench verified*, 2024. [Online]. Available: `https://openai.com/index/introducing-swe-bench-verified/`

[100] J. Yang et al., “SWE-agent: Agent-computer interfaces enable automated software engineering,” in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. [Online]. Available: `https://openreview.net/forum?id=mXpq6ut8J3`

[101] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, “Autocoderover: Autonomous program improvement,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024, Vienna, Austria: Association for Computing Machinery, 2024, pp. 1592–1604, ISBN: 9798400706127. DOI: `10.1145/3650212.3680384` [Online]. Available: `https://doi.org/10.1145/3650212.3680384`

[102] H. Ruan, Y. Zhang, and A. Roychoudhury, “Specrover: Code intent extraction via llms,” in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ser. ICSE ’25, Ottawa, Ontario, Canada: IEEE Press, 2025, pp. 963–974, ISBN: 9798331505691. DOI: 10.1109/ICSE55347.2025.00080 [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00080

[103] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, “Demystifying llm-based software engineering agents,” *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025. DOI: 10.1145/3715754 [Online]. Available: https://doi.org/10.1145/3715754

[104] X. Wang et al., “Openhands: An open platform for AI software developers as generalist agents,” in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: https://openreview.net/forum?id=OJd3ayDDoF

[105] I. Bouzenia, P. Devanbu, and M. Pradel, “Repairagent: An autonomous, llm-based agent for program repair,” in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ser. ICSE ’25, Ottawa, Ontario, Canada: IEEE Press, 2025, pp. 2188–2200, ISBN: 9798331505691. DOI: 10.1109/ICSE55347.2025.00157 [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00157

[106] Y. Wei et al., “Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution,” in *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. [Online]. Available: https://openreview.net/forum?id=ULb1O61XZ0

[107] Amazon, *Amazon q developer*, 2026. [Online]. Available: https://aws.amazon.com/q/developer/

[108] Cognition, *Introducing devin, the first ai software engineer*, 2026. [Online]. Available: https://cognition.ai/blog/introducing-devin

[109] A. Khatry et al., “CRUST-bench: A comprehensive benchmark for c-to-safe-rust transpilation,” in *Second Conference on Language Modeling*, 2025. [Online]. Available: https://openreview.net/forum?id=8xofWL61S9

[110] Z. Guan, X. Yin, Z. Peng, and C. Ni, “Repotransagent: Multi-agent llm framework for repository-aware code translation,” *arXiv preprint arXiv:2508.17720*, 2025.

[111] T. Li, R. Li, B. Wang, B. Paulsen, U. Mathur, and P. Saxena, “Adversarial agent collaboration for c to rust translation,” *arXiv preprint arXiv:2510.03879*, 2025.

[112] H. Sim, H. Cho, Y. Go, Z. Fu, A. Shokri, and B. Ravindran, “Large language model-powered agent for c to rust code translation,” *arXiv preprint arXiv:2505.15858*, 2025.

[113] Tree-Sitter, *Tree-sitter library*, 2026. [Online]. Available: https://tree-sitter.github.io/tree-sitter/

[114] L. A. Agrawal, A. Kanade, N. Goyal, S. Lahiri, and S. Rajamani, “Monitor-guided decoding of code lms with static analysis of repository context,” in *Advances in Neural Information Processing Systems*, vol. 36, 2023, pp. 32 270–32 298. [Online]. Available: https://neurips.cc/media/neurips-2023/Slides/70362.pdf

[115] N. McAleese, R. M. Pokorny, J. F. C. Uribe, E. Nitishinskaya, M. Trebacz, and J. Leike, “LLM Critics Help Catch LLM Bugs,” *arXiv preprint arXiv:2407.00215*, 2024.

[116] Z. Lin, S. Shen, J. Shang, J. E. Weston, and Y. Nie, “Learning to solve and verify: A self-play framework for mutually improving code and test generation,” in *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, 2025. [Online]. Available: https://openreview.net/forum?id=j6tMZaPWWF

[117] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, and H. Cui, “Agentcoder: Multi-agent-based code generation with iterative testing and optimisation,” *arXiv preprint arXiv:2312.13010*, 2023.

[118] C. Qian et al., “ChatDev: Communicative Agents for Software Development,” in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 15 174–15 186.

[119] Y. Dong, X. Jiang, Z. Jin, and G. Li, “Self-collaboration code generation via chatgpt,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–38, 2024.

[120] M. A. Islam, M. E. Ali, and M. R. Parvez, “MapCoder: Multi-Agent Code Generation for Competitive Problem Solving,” in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds., Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 4912–4944. DOI: `10.18653/v1/2024.acl-long.269` [Online]. Available: `https://aclanthology.org/2024.acl-long.269/`

[121] J. Karwowski, O. Hayman, X. Bai, K. Kiendlhofer, C. Griffin, and J. M. V. Skalse, “Goodhart’s law in reinforcement learning,” in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: `https://openreview.net/forum?id=5o9G4XF1LI`

[122] A. Pan, E. Jones, M. Jagadeesan, and J. Steinhardt, “Feedback loops with language models drive in-context reward hacking,” in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML’24, Vienna, Austria: JMLR.org, 2024.

[123] Y. Liu, N. S. Moosavi, and C. Lin, “Llms as narcissistic evaluators: When ego inflates evaluation scores,” in *Findings of the Association for Computational Linguistics: ACL 2024*, 2024, pp. 12 688–12 701.

[124] H. Li et al., “Llms-as-judges: A comprehensive survey on llm-based evaluation methods,” *arXiv preprint arXiv:2412.05579*, 2024.

[125] D. Roytburg, M. Bozoukov, M. Nguyen, J. Barzdukas, M. Puig-Hall, and N. Oozeer, "Are llm evaluators really narcissists? sanity checking self-preference evaluations," *arXiv preprint arXiv:2601.22548*, 2026.

[126] S. Ren et al., "Codebleu: A method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[127] A. Eghbali and M. Pradel, "Crystalbleu: Precisely and efficiently measuring the similarity of code," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[128] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang, "AVATAR: A parallel corpus for Java-python program translation," in *Findings of the Association for Computational Linguistics: ACL 2023*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds., Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 2268–2281. DOI: `10.18653/v1/2023.findings-acl.143` [Online]. Available: `https://aclanthology.org/2023.findings-acl.143/`

[129] M. Chen et al., "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[130] R. Puri et al., "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[131] B. Beizer, *Software testing techniques*, 1990.

[132] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.

[133] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 129–140.

[134] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1135–1146.

[135] S. Ouyang et al., "Repograph: Enhancing AI software engineering with repository-level code graph," in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: `https://openreview.net/forum?id=dw9VUsSHGB`

[136] P. Chen and G. Lampouras, "Exploring data augmentation for code generation tasks," in *Findings of the Association for Computational Linguistics: EACL 2023*, A. Vlachos and I. Augenstein, Eds., Dubrovnik, Croatia: Association for Computational Linguistics, May 2023, pp. 1542–1550. DOI: `10.18653/v1/2023.findings-eacl.114` [Online]. Available: `https://aclanthology.org/2023.findings-eacl.114/`

[137] D. Heo and H. Choi, "End-to-end training of back-translation framework with categorical reparameterization trick," in *Artificial Neural Networks and Machine Learning – ICANN 2024: 33rd International Conference on Artificial Neural Networks, Lugano, Switzerland, September 17–20, 2024, Proceedings, Part VII*, Lugano, Switzerland: Springer-Verlag, 2024, pp. 177–193, ISBN: 978-3-031-72349-0. DOI: `10.1007/978-3-031-72350-6_12` [Online]. Available: `https://doi.org/10.1007/978-3-031-72350-6_12`

[138] I. M. Subedi, M. Singh, V. Ramasamy, and G. S. Walia, "Application of back-translation: A transfer learning approach to identify ambiguous software

requirements," in *Proceedings of the 2021 ACM Southeast Conference*, 2021, pp. 130–137.

[139] A. Silva, J. F. Ferreira, H. Ye, and M. Monperrus, "Mufin: Improving neural repair models with back-translation," *arXiv preprint arXiv:2304.02301*, 2023.

[140] *Will code move on to a language such as rust?* `https://www.theregister.com/2020/06/30/hard_to_find_linux_maintainers_says_torvalds/`, 2020.

[141] *Supporting linux kernel development in rust*, `https://lwn.net/Articles/829858/`, 2020.

[142] *Upgrading github from rails 3.2 to 5.2*, `https://github.blog/2018-09-28-upgrading-github-from-rails-3-2-to-5-2/`, 2018.

[143] *Github's journey from monolith to microservices*, `https://www.infoq.com/articles/github-monolith-microservices/`, 2021.

[144] *Transform monolithic java applications into microservices with the power of ai*, `https://developer.ibm.com/tutorials/transform-monolithic-java-applications-into-microservices-with-the-power-of-ai/`, 2020.

[145] F. Liu, J. Li, and L. Zhang, "Syntax and domain aware model for unsupervised program translation," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23, Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 755–767, ISBN: 9781665457019. DOI: `10.1109/ICSE48619.2023.00072` [Online]. Available: `https://doi.org/10.1109/ICSE48619.2023.00072`

[146] M. Szafraniec, B. Roziere, H. J. Leather, P. Labatut, F. Charton, and G. Synnaeve, "Code translation with compiler representations," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: `https://openreview.net/forum?id=XomEU3eNeSQ`

[147] Y. Wen et al., "Babeltower: Learning to auto-parallelized program translation," in *International Conference on Machine Learning*, PMLR, 2022, pp. 23 685–23 700.

[148] C.-K. Ting, K. Munson, S. Wade, A. Savla, K. Kate, and K. Srinivas, "Codestylist: A system for performing code style transfer using neural networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, 2023, pp. 16 485–16 487.

[149] J. Hong, "Improving automatic c-to-rust translation with static analysis," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, IEEE, 2023, pp. 273–277.

[150] J. D. Weisz et al., "Perfection not required? human-ai partnerships in code translation," in *26th International Conference on Intelligent User Interfaces*, 2021, pp. 402–412.

[151] J. D. Weisz et al., "Better together? an evaluation of ai-supported code translation," in *27th International Conference on Intelligent User Interfaces*, 2022, pp. 369–391.

[152] J. White et al., "A prompt pattern catalog to enhance prompt engineering with chatgpt," in *Proceedings of the 30th Conference on Pattern Languages of Programs*, ser. PLoP '23, Monticello, IL, USA: The Hillside Group, 2023, ISBN: 9781941652190.

[153] S. Abukhalaf, M. Hamdaqa, and F. Khomh, "On codex prompt engineering for ocl generation: An empirical study," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 148–157. DOI: 10.1109/MSR59073.2023.00033

[154] X. Jiang et al., "Self-planning code generation with large language models," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 7, Sep. 2024, ISSN: 1049-331X. DOI: 10.1145/3672456 [Online]. Available: https://doi.org/10.1145/3672456

[155] T. Y. Zhuo et al., "On robustness of prompt-based semantic parsing with large pre-trained language model: An empirical study on codex," in *Proceedings of the 17th Conference of the European Chapter of the Association for Computational

*Linguistics*, A. Vlachos and I. Augenstein, Eds., Dubrovnik, Croatia: Association for Computational Linguistics, May 2023, pp. 1090–1102. DOI: 10.18653/v1/2023.eacl-main.77 [Online]. Available: https://aclanthology.org/2023.eacl-main.77/

[156] *Hugging face open llm leaderboard*, https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard, 2023.

[157] *Thebloke wizard vicuna 13b*, https://huggingface.co/TheBloke/Wizard-Vicuna-13B-Uncensored-HF, 2023.

[158] *Thebloke airoboros 13b*, https://huggingface.co/TheBloke/airoboros-13B-HF, 2023.

[159] TIOBE, *Tiobe index*, https://www.tiobe.com/tiobe-index, 2024.

[160] *Codegeex*, https://github.com/THUDM/CodeGeeX/blob/main/tests/test_prompt.txt, 2023.

[161] *Py_compile—compile python source files*, https://docs.python.org/3/library/py_compile.html, 2023.

[162] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Chi conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.

[163] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[164] S. Yao et al., “Tree of thoughts: Deliberate problem solving with large language models,” in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS ’23, New Orleans, LA, USA: Curran Associates Inc., 2023.

[165] J. Howard and S. Ruder, “Universal language model fine-tuning for text classification,” in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, I. Gurevych and Y. Miyao, Eds., Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 328–339. DOI: `10.18653/v1/P18-1031` [Online]. Available: `https://aclanthology.org/P18-1031/`

[166] C. Sun, X. Qiu, Y. Xu, and X. Huang, “How to fine-tune bert for text classification?” In *Chinese Computational Linguistics: 18th China National Conference, CCL 2019, Kunming, China, October 18–20, 2019, Proceedings 18*, Springer, 2019, pp. 194–206.

[167] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, “Adaptive mixtures of local experts,” *Neural computation*, vol. 3, no. 1, pp. 79–87, 1991.

[168] `https://sourceforge.net/projects/j2cstranslator`, 2023.

[169] *Gpt-4 technical report*, `https://cdn.openai.com/papers/gpt-4.pdf`, 2023.

[170] M. Khan et al., “Modernization framework to enhance the security of legacy information systems,” *Intelligent Automation & Soft Computing*, vol. 32, no. 1, pp. 543–555, 2022, ISSN: 2326-005X. DOI: `10.32604/iasc.2022.016120` [Online]. Available: `http://www.techscience.com/iasc/v32n1/45267`

[171] A. Terekhov and C. Verhoef, “The realities of language conversions,” *IEEE Software*, vol. 17, no. 6, pp. 111–124, 2000. DOI: `10.1109/52.895180`

[172] G. Guizzo, J. M. Zhang, F. Sarro, C. Treude, and M. Harman, "Mutation analysis for evaluating code translation," *Empirical Software Engineering*, vol. 29, no. 1, p. 19, Dec. 2023, ISSN: 1573-7616. DOI: 10.1007/s10664-023-10385-w [Online]. Available: https://doi.org/10.1007/s10664-023-10385-w

[173] L. Qiu, "Programming language translation," Cornell University, USA, Tech. Rep., 1999.

[174] D. Goerge, P. Girase, M. Gupta, P. Gupta, and A. Sharma, "Programming language inter-conversion," *International Journal of Computer Applications*, vol. 1, no. 20, pp. 63–69, Feb. 2010, ISSN: 0975-8887. DOI: 10.5120/419-619 [Online]. Available: https://ijcaonline.org/archives/volume1/number20/419-619/

[175] D. Broggi and Y. Liu, "On the interoperability of programming languages via translation," in *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)*, 2023, pp. 2579–2585. DOI: 10.1109/CSCE60160.2023.00413

[176] H. A. O. Eman J. Coco and N. I. Osman, "Jpt : A simple java-python translator," *CAIJ*, vol. 5, no. 2, pp. 1–18, 2018. DOI: 10.5121/caij.2018.5201

[177] K. Lano and H. Siala, "Using model-driven engineering to automate software language translation," *Automated Software Engineering*, vol. 31, no. 1, p. 20, Feb. 2024, ISSN: 1573-7535. DOI: 10.1007/s10515-024-00419-y [Online]. Available: https://doi.org/10.1007/s10515-024-00419-y

[178] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20, Vancouver, BC, Canada: Curran Associates Inc., 2020, ISBN: 9781713829546.

[179] T. Würthinger et al., "One vm to rule them all," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on*

*Programming & Software*, ser. Onward! 2013, Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 187–204, ISBN: 9781450324724. DOI: 10.1145/2509578.2509581 [Online]. Available: https://doi.org/10.1145/2509578.2509581

[180] GraalVM, *Polyglot api*, https://www.graalvm.org/latest/reference-manual/polyglot-programming, 2024.

[181] GitHub, *Codeql*, 2026. [Online]. Available: https://codeql.github.com

[182] T. J. Team, *Junit*, 2024. [Online]. Available: https://junit.org/junit5/

[183] pytest-dev, *Pytest*, https://www.pytest.org, 2024.

[184] T. J. Team, *Java code coverage*, 2024. [Online]. Available: https://www.eclemma.org/jacoco/

[185] N. Batchelder, *Coverage.py*, https://pypi.org/project/coverage, 2024.

[186] T. A. S. Foundation, *Apache commons cli*, 2026. [Online]. Available: https://github.com/apache/commons-cli

[187] T. A. S. Foundation, *Apache commons codec*, 2024. [Online]. Available: https://github.com/apache/commons-codec

[188] T. A. S. Foundation, *Apache commons csv*, 2026. [Online]. Available: https://github.com/apache/commons-csv

[189] T. A. S. Foundation, *Apache commons exec*, 2024. [Online]. Available: https://github.com/apache/commons-exec

[190] D. Lemire, *Javafastpfor*, 2024. [Online]. Available: https://github.com/lemire/JavaFastPFOR

[191] T. A. S. Foundation, *Apache commons fileupload*, 2026. [Online]. Available: https://github.com/apache/commons-fileupload

[192] T. A. S. Foundation, *Apache commons graph*, 2024. [Online]. Available: `https://github.com/apache/commons-graph`

[193] F. Source, *Jansi*, 2024. [Online]. Available: `https://github.com/fusesource/jansi`

[194] T. A. S. Foundation, *Apache commons pool*, 2024. [Online]. Available: `https://github.com/apache/commons-pool`

[195] T. A. S. Foundation, *Apache commons validator*, 2026. [Online]. Available: `https://github.com/apache/commons-validator`

[196] C. Spearman, "The proof and measurement of association between two things.," *The American Journal of Psychology*, vol. 15, no. 1, pp. 72–101, 1904. DOI: `10.2307/1412159` [Online]. Available: `https://doi.org/10.2307/1412159`

[197] *Pylint static code analyzer for python*, 2024. [Online]. Available: `https://pypi.org/project/pylint/`

[198] *Black: Python code formatter*, 2024. [Online]. Available: `https://github.com/psf/black`

[199] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11, Szeged, Hungary: Association for Computing Machinery, 2011, pp. 416–419, ISBN: 9781450304436. DOI: `10.1145/2025113.2025179` [Online]. Available: `https://doi.org/10.1145/2025113.2025179`

[200] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. Tulio Valente, "How do developers react to api evolution? the pharo ecosystem case," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 251–260. DOI: `10.1109/ICSM.2015.7332471`

[201] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, “Do developers update their library dependencies?” *Empirical Softw. Engg.*, vol. 23, no. 1, pp. 384–417, Feb. 2018, ISSN: 1382-3256. DOI: 10.1007/s10664-017-9521-5 [Online]. Available: https://doi.org/10.1007/s10664-017-9521-5

[202] Y. Wang et al., “An empirical study of usages, updates and risks of third-party libraries in java projects,” in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 35–45. DOI: 10.1109/ICSME46990.2020.00014

[203] A. Wei et al., “EquiBench: Benchmarking large language models’ reasoning about program semantics via equivalence checking,” in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, C. Christodoulopoulos, T. Chakraborty, C. Rose, and V. Peng, Eds., Suzhou, China: Association for Computational Linguistics, Nov. 2025, pp. 33 868–33 881, ISBN: 979-8-89176-332-6. DOI: 10.18653/v1/2025.emnlp-main.1718 [Online]. Available: https://aclanthology.org/2025.emnlp-main.1718/

[204] N. Maveli, A. Vergari, and S. B. Cohen, “What can large language models capture about code functional equivalence?” In *Findings of the Association for Computational Linguistics: NAACL 2025*, L. Chiruzzo, A. Ritter, and L. Wang, Eds., Albuquerque, New Mexico: Association for Computational Linguistics, Apr. 2025, pp. 6865–6903, ISBN: 979-8-89176-195-7. DOI: 10.18653/v1/2025.findings-naacl.382 [Online]. Available: https://aclanthology.org/2025.findings-naacl.382/

[205] Anthropic, *Building effective ai agents*, https://www.anthropic.com/engineering/building-effective-agents, 2025.

[206] T. C. C. Team, *Claude code*, 2026. [Online]. Available: https://github.com/anthropics/claude-code

[207] T. O. Team, *Openai codex cli*, 2026. [Online]. Available: https://github.com/openai/codex

[208] S.-H. Cha, "Comprehensive survey on distance/similarity measures between probability density functions," *City*, vol. 1, no. 2, p. 1, 2007.

[209] F. P. Miller, A. F. Vandome, and J. McBrewster, *Levenshtein Distance: Information theory, Computer science, String (computer science), String metric, Damerau?Levenshtein distance, Spell checker, Hamming distance*. Alpha Press, 2009, ISBN: 6130216904.

[210] O. Tonomori, *Checkdigit*, 2026. [Online]. Available: https://github.com/osamingo/checkdigit

[211] H. Bollon, *Go-edlib : Edit distance and string comparison library*, 2026. [Online]. Available: https://github.com/hbollon/go-edlib

[212] V. Cortex, *Gohistogram - histograms in go*, 2026. [Online]. Available: https://github.com/VividCortex/gohistogram

[213] J. Polera, *Gonameparts*, 2026. [Online]. Available: https://github.com/polera/gonameparts

[214] M. Flynn, *Stats - golang statistics package*, 2026. [Online]. Available: https://github.com/montanaflynn/stats

[215] D. Belicza, *Textrank on go*, 2026. [Online]. Available: https://github.com/DavidBelicza/TextRank

[216] T. Algorithms, *All algorithms implemented in python*, 2026. [Online]. Available: https://github.com/TheAlgorithms/Python/blob/master/data_structures/binary_tree/binary_search_tree_recursive.py

[217] T. P. Team, *Conversion functions between rgb and other color systems*, 2026. [Online]. Available: `https://github.com/python/cpython/blob/3.13/Lib/colorsys.py`

[218] T. P. Team, *Heap queue algorithm (a.k.a. priority queue)*, 2026. [Online]. Available: `https://github.com/python/cpython/blob/3.13/Lib/heapq.py`

[219] T. P. Team, *A parser for html and xhtml*, 2026. [Online]. Available: `https://github.com/python/cpython/blob/3.13/Lib/html/parser.py`

[220] L. Weiler, *Basic math*, 2026. [Online]. Available: `https://github.com/lukew3/mathgenerator/blob/main/mathgenerator/basic_math.py`

[221] T. Algorithms, *All algorithms implemented in python*, 2026. [Online]. Available: `https://github.com/TheAlgorithms/Python/blob/master/data_structures/binary_tree/red_black_tree.py`

[222] Z. Luo, *A library implementing different string similarity and distance measures using python*, 2026. [Online]. Available: `https://github.com/luozhouyang/python-string-similarity/tree/master/strsimpy`

[223] W. Pearson, *Python lib for toml*, 2026. [Online]. Available: `https://github.com/uiri/toml/tree/master/toml`

[224] Jawah, *Charset normalizer: Truly universal encoding detector in pure python*, 2025. [Online]. Available: `https://github.com/jawah/charset_normalizer`

[225] D. Chat, *Delta.chat c-library with e2e chat-over-email functionality & python bindings*, 2025. [Online]. Available: `https://github.com/deltachat/deltachat-core`

[226] T. A. S. Foundation, *Apache iceberg*, 2025. [Online]. Available: `https://github.com/apache/iceberg`

[227] T. A. S. Foundation, *Apache pyiceberg*, 2025. [Online]. Available: https://github.com/apache/iceberg-python

[228] T. A. S. Foundation, *Amcl - apache milagro crypto library*, 2025. [Online]. Available: https://github.com/apache/incubator-milagro-crypto-c

[229] T. A. S. Foundation, *Mcjl - milagro crypto java library*, 2025. [Online]. Available: https://github.com/apache/incubator-milagro-java

[230] T. P. Team, *Cpython*, https://github.com/python/cpython, 2025.

[231] libp2p, *The python implementation of the libp2p networking stack*, 2025. [Online]. Available: https://github.com/libp2p/py-libp2p

[232] T. S.-b. Team, *Swe-bench leaderboard*, 2026. [Online]. Available: https://www.swebench.com/

[233] T. B. Team, *Bigcodebench leaderboard*, 2026. [Online]. Available: https://bigcode-bench.github.io/

[234] T. O. Team, *Gpt-4o*, 2025. [Online]. Available: https://openai.com/index/hello-gpt-4o/

[235] T. G. D. Team, *Gemini pro*, 2026. [Online]. Available: https://deepmind.google/models/gemini/pro/

[236] OpenAI, *Introducing openai o3 and o4-mini*, https://openai.com/index/introducing-o3-and-o4-mini/, 2026.

[237] T. R. L. Team, *Rust language*, 2026. [Online]. Available: https://www.rust-lang.org/

[238] T. P. L. Team, *Python language*, 2026. [Online]. Available: https://www.python.org/

[239] T. J. L. Team, *Java language*, 2026. [Online]. Available: https://www.java.com/en/

[240] T. N. Team, *Nodejs*, 2026. [Online]. Available: https://nodejs.org/en

[241] T. G. Team, *C compiler*, 2026. [Online]. Available: https://gcc.gnu.org/

[242] T. G. L. Team, *Go language*, 2026. [Online]. Available: https://go.dev/

[243] L. Team, *Litellm*, 2026. [Online]. Available: https://www.litellm.ai/

[244] C. Ziftci, S. Nikolov, A. Sjövall, B. Kim, D. Codecasa, and M. Kim, "Migrating code at scale with llms at google," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, ser. FSE Companion '25, Clarion Hotel Trondheim, Trondheim, Norway: Association for Computing Machinery, 2025, pp. 162–173, ISBN: 9798400712760. DOI: 10.1145/3696630.3728542 [Online]. Available: https://doi.org/10.1145/3696630.3728542

[245] J. J. Garzella, M. Baranowski, S. He, and Z. Rakamarić, "Leveraging compiler intermediate representation for multi- and cross-language verification," in *Verification, Model Checking, and Abstract Interpretation: 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16–21, 2020, Proceedings*, New Orleans, LA, USA: Springer-Verlag, 2020, pp. 90–111, ISBN: 978-3-030-39321-2. DOI: 10.1007/978-3-030-39322-9_5 [Online]. Available: https://doi.org/10.1007/978-3-030-39322-9_5

[246] J. Liu et al., "Large language model-based agents for software engineering: A survey," *ACM Trans. Softw. Eng. Methodol.*, Mar. 2026, Just Accepted, ISSN: 1049-331X. DOI: 10.1145/3796507 [Online]. Available: https://doi.org/10.1145/3796507

[247] Z. Xi et al., "The rise and potential of large language model based agents: A survey," *Science China Information Sciences*, vol. 68, no. 2, p. 121 101, Jan. 2025, ISSN: 1869-1919. DOI: 10.1007/s11432-024-4222-0 [Online]. Available: https://doi.org/10.1007/s11432-024-4222-0

[248] T. M. T. Team, *Moatless tools*, 2026. [Online]. Available: https://github.com/aorwall/moatless-tools

[249] A. AI, *Ai pair programming in your terminal*, 2026. [Online]. Available: https://aider.chat/

[250] L. E. Erdogan et al., "Plan-and-act: Improving planning of agents for long-horizon tasks," in *Forty-second International Conference on Machine Learning*, 2025. [Online]. Available: https://openreview.net/forum?id=ybA4EcMmUZ

[251] K. Chen et al., "Reinforcement learning for long-horizon interactive llm agents," *arXiv preprint arXiv:2502.01600*, 2025.

[252] W. Sun et al., "Scaling long-horizon llm agent via context-folding," *arXiv preprint arXiv:2510.11967*, 2025.

[253] T. R. L. Team, *Rust analyzer*, 2026. [Online]. Available: https://rust-analyzer.github.io/

[254] T. L. Team, *Clangd*, 2026. [Online]. Available: https://github.com/clangd/clangd

[255] T. G. Team, *Gopls: The language server for go*, 2026. [Online]. Available: https://go.dev/gopls/

[256] T. T. L. S. Team, *Typescript language server*, 2026. [Online]. Available: https://github.com/typescript-language-server/typescript-language-server

[257] T. S. I. Team, *Python lsp server*, 2026. [Online]. Available: https://github.com/python-lsp/python-lsp-server

[258] T. E. Team, *Eclipse jdt language server*, 2026. [Online]. Available: https://github.com/eclipse-jdtls/eclipse.jdt.ls

[259] Microsoft, *Language server implementations*, 2026. [Online]. Available: https://microsoft.github.io/language-server-protocol/implementors/servers/

[260] Y. Wang et al., "Repotransbench: A real-world multilingual benchmark for repository-level code translation," *IEEE Transactions on Software Engineering*, 2025.

[261] J. Yang et al., "Swe-agent: Agent-computer interfaces enable automated software engineering," *Advances in Neural Information Processing Systems*, vol. 37, pp. 50 528–50 652, 2024.

[262] T. Q. Team, *Qwen embedding*, 2026. [Online]. Available: `https://huggingface.co/Qwen/Qwen3-Embedding-0.6B`