ReCodeAgent: A Multi-agent Workflow for Language-Agnostic Translation and Validation of Large-Scale Repositories

Ali Reza Ibrahimzada
University of Illinois Urbana-Champaign
Urbana, IL, USA
alirezai@illinois.edu

Brandon Paulsen
Amazon
Arlington, VA, USA
bpaulse@amazon.com

Daniel Kroening
Amazon
Seattle, WA, USA
dkr@amazon.co.uk

Reyhaneh Jabbarvand
University of Illinois Urbana-Champaign
Urbana, IL, USA
reyhaneh@illinois.edu

Abstract

Most repository-level code translation and validation techniques have been evaluated on a single source-target programming language (PL) pair, owing to the complex engineering effort required to adapt new PL pairs. *Programming agents* can enable PL-agnosticism in repository-level code translation and validation: they can synthesize code across many PLs and autonomously use existing tools specific to each PL's analysis. However, state-of-the-art has yet to offer a fully autonomous agentic approach for *repository-level* code translation and validation of *large-scale* programs. This paper proposes ReCodeAgent, an autonomous multi-agent approach for language-agnostic repository-level code translation and validation. Users only need to provide the project in the source PL and specify the target PL for ReCodeAgent to automatically translate and validate the entire repository. ReCodeAgent is the *first* technique to achieve high translation success rates across many PLs.

We compare the effectiveness of ReCodeAgent with four alternative neuro-symbolic and agentic approaches to translate 118 real-world projects, with 1,975 LoC and 43 translation units for each project, on average. The projects cover 6 PLs and 4 PL pairs. Our results demonstrate that ReCodeAgent consistently outperforms prior techniques on translation correctness, improving test pass rate by 60.8% on ground-truth tests, with an average cost of $15.3. We also perform process-centric analysis of ReCodeAgent trajectories to confirm its procedural efficiency. Finally, we investigate how the design choices (a multi-agent vs. single-agent architecture) influence ReCodeAgent performance: on average, the test pass rate drops by 40.4%, and trajectories become 28% longer and persistently inefficient.

CCS Concepts

• **Software and its engineering → Source code generation**.

Keywords

Code Translation and Validation, Programming Agents



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837486

ACM Reference Format:
Ali Reza Ibrahimzada, Brandon Paulsen, Daniel Kroening, and Reyhaneh Jabbarvand. 2026. ReCodeAgent: A Multi-agent Workflow for Language-Agnostic Translation and Validation of Large-Scale Repositories. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837486

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 [23, 24, 28, 29]. Early work developed rule-based approaches [20, 21, 46, 63], like C2Rust, where all translation rules are written by hand. Later work developed neuro-symbolic techniques [5, 11, 18, 36, 43, 51, 65–67, 79–81], which combine large language models (LLMs) with (symbolic) program analysis and testing. More recently, agentic approaches have been evaluated [16, 30, 31, 52], wherein one or more LLM agents work together to translate code between specific source-target PLs.

Prior rule-based and neuro-symbolic translation techniques only evaluate on a single source-target PL pair [11, 18, 20, 43, 51, 65, 70, 76, 80]. 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[1] and neuro-symbolic tools over 10K LoC[2] 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 [77]. 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

[1] C2Rust [20] (100K+ LoC).
[2] AlphaTrans [18] (11K LoC), Oxidizer [80] (19K LoC), and Skel [65] (4K LoC).

only explore the codebase [30] or use PL-specific tools [31]. For example, Crust's SWE-agent uses only file navigation and build/test commands and underperforms a generate-then-repair loop [30]; ACToR depends on C/Rust CLI differential fuzzing [31]; and RepoTransAgent's tools are hardcoded for Java–C# lookups [16].

(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 [6, 14, 53], 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. Long-horizon agents lose high-level goals without an explicit plan [14], confabulate under multi-step tool feedback [6], and degrade as interaction history grows unbounded [53]; in repository translation, AlphaTrans shows LLMs hallucinate types (including non-existent ones), method names, and assert APIs [18].

(3) Dichotomy of Test Translation. Translating first and validating next approach simply does not work for large-scale repository-level translation because of long call chains and test coupling effect [18]. In AlphaTrans, each test averages 3 direct callees and 27 executed methods, so one buggy callee blocks validating the rest [18]. 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 [13, 17, 22, 32, 37, 47]; e.g., co-generating code and tests biases the tests [17]. Otherwise, the agent may modify the test rather than fix the incorrect code, e.g., by removing or relaxing assertions—Ahmed et al. [3] reports 21.8%–33.0% passing tests generated but fail golden tests in SWE-bench. At the same time, test translation in a real-world setting is known to be even more challenging than code translation [1, 45], 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. Prior agentic techniques rarely discuss the principled design space of an agentic code translation workflow [16, 30, 31]. 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 [18], 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 [27]. There is also a dearth of transparency in cost reporting, and no attempts to analyze trajectories beyond final translation outcomes. Outcome-centric metrics alone are insufficient: on Crust, SWE-agent reaches the same test pass rate as a generate-repair loop while wasting many steps on file navigation [30]; ACToR and RepoTransAgent report primarily pass/compile rates [16, 31]. Test outcomes can also mislead: AlphaTrans notes that automatically generated tests can raise pass rates with weak assertions [18], TRAM measures 2.4%/7.6% false positives/negatives under GraalVM-based validation [27].

This paper presents ReCodeAgent, a multi-agent framework for *language-agnostic*, repository-level code translation and validation (§3). ReCodeAgent leverages four specialized agents, dividing the overall task into distinct phases: analysis, planning, translation, and validation. The Analyzer Agent (§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 (§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 [15]) with tool-assisted, LLM-centered static analysis (lightweight tools that support many PLs, e.g., Tree-sitter [64]). The plan guides subsequent agents with concrete steps to avoid unnecessary exploration, which can cause hallucinations (*challenge 2*). The Translator agent (§3.4) carries out the steps outlined in the plan to co-translate code and tests. The Validator agent (§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. [16] and Li et al. [31].

We evaluate the effectiveness of ReCodeAgent against four existing neuro-symbolic and agentic techniques [18, 30, 65, 80]. Our benchmark comprises 4,583 translation units, drawn from 118 real-world projects totaling over 230*K* LoC (§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 (§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 (§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 [34]. 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 (§4.4). ReCodeAgent is cost-effective, translating and validating projects in 57 minutes with a cost of $15.3, on average (§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

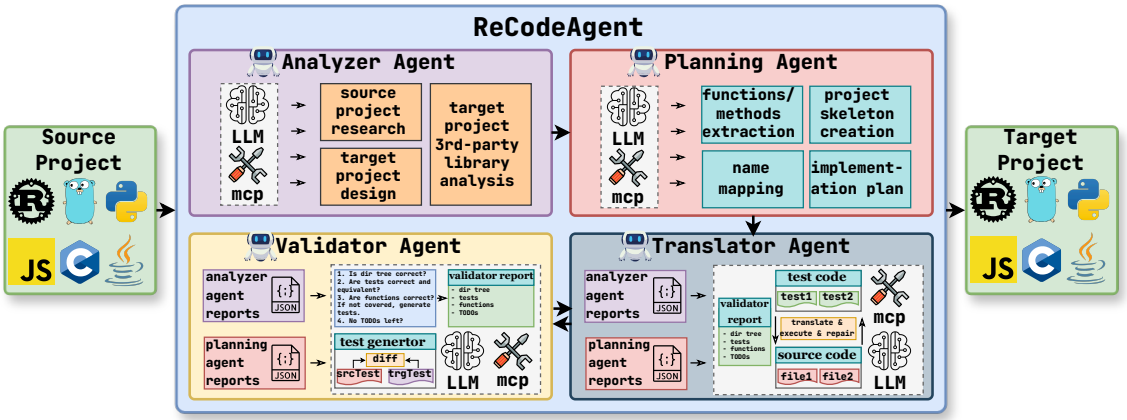


Figure 1: Overview of ReCodeAgent.

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 [48].

2 Problem Definition and Architecture Design

A source project $P_s = (F_s, T_s, D_s)$ consists of source functions $F_s = \{f_s^1, \ldots, f_s^n\}$, 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 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 1 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* (§3.2), *Planning Agent* (§3.3), *Translator Agent* (§3.4), and *Validator Agent* (§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 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.

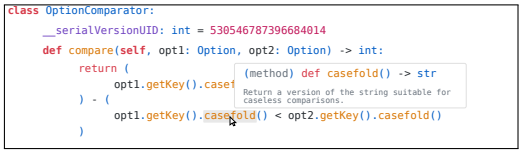


Figure 2: Hover feature on a Python code in an IDE.

```
==================================== 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 3: Hover tool from the Python LSP server.

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.

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 (§3.1) and explain the details and workflow of each agent for code translation and validation through reasoning and tool usage (§3.2–§3.5).

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.

3.1.1 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 [2]. 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 [55–57, 59–61] 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 [38], without writing any additional code. LSP functionalities that we integrate 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 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 2 shows the hover feature in a normal IDE, while the same functionality from the LSP server is given in Figure 3.

(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.

3.1.2 Project Analysis (PA) Tools: These tools extract structural information from the 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 4 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.

3.2 Analyzer Agent

The *Analyzer Agent* conducts initial research and formulates the high-level design of the translation (Algorithm 1, line 22). This

```
======== 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 4: Project analysis (PA) tools output.

agent ensures that the target project preserves structurally similar to the source, and identifies the most suitable libraries and design patterns for the target PL. Figure 5 illustrates the three documents produced by this agent, corresponding to the following phases:

3.2.1 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.

3.2.2 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.

3.2.3 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* (§3.3), *Translator* (§3.4), and *Validator* (§3.5) *Agents*.

3.3 Planning Agent

The *Planning Agent* reads the source project research and target project design documents generated by the *Analyzer Agent* (§3.2) and decomposes the high-level design into granular, executable implementation steps (Algorithm 1, line 23). This agent ensures that every source code file is translated and validated according to a logical, dependency-aware order. Figure 6 illustrates the documents and skeleton files produced by the planning agent.

3.3.1 Fragment Extraction. The agent first extracts all translation units—including functions, methods, and classes—from available

```
============ 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 5: Documents generated by *Analyzer Agent* in ReCodeAgent.

```
======== Fragment Extraction =======
## checkdigit.go
   checkdigit.go:isNumber
   checkdigit.go:NewLuhn
   checkdigit.go:NewDamm
   checkdigit.go:NewUPC

## damm.go
```

```
=========== Name Mapping ===========
## functions:
   go.isNumber: rs.isNumber
   go.NewLuhn: rs.NewLuhn
   go.NewDamm: rs.NewDamm
   go.NewUPC: rs.NewUPC
## variables:
   go. ...
```

```
======== Skeleton Generation =======
fn isNumber(n: char) -> bool {}

fn NewLuhn() -> impl Pvd {}

fn NewDamm() -> impl Pvd {}

fn NewUPC() -> impl Pvd {}
```

```
======== Implementation Plan =======
## Overview
## Part A:
   A1: Translate checkdigit.go
   A2: Translate damm.go
## Part B:
   B1: Translate cd_test.go
   B2: Translate damm_test.go
```

Figure 6: Documents generated by *Planning Agent* in ReCodeAgent.

Algorithm 1: ReCodeAgent

```
   Input  : sourceProject, LLM, tools, timeout = 5000s, maxIter = 5, validationReport ← ∅
   Output : translatedValidatedProject
1  Function runTranslatorAgent (context, validationReport):
2      translatorAgent ← initializeAgent (LLM, tools, context)
3      if validationReport ≠ ∅ then
4          return translatorAgent.repair (validationReport)
5      implementationPlan ← context.planningOutput.plan
6      translatedProject ← ∅
7      foreach Part-A in implementationPlan do
8          agentTranslatedSourceCode ← translatorAgent.implement (Part-A)
9          translatedProject ← translatedProject ∪ agentTranslatedSourceCode
10     foreach Part-B in implementationPlan do
11         agentTranslatedTestCode ← translatorAgent.implement (Part-B)
12         translatedProject ← translatedProject ∪ agentTranslatedTestCode
13     return translatedProject
14 Function runValidatorAgent (context, translatedProject):
15     validatorAgent ← initializeAgent (LLM, tools, context)
16     validationReport ← validatorAgent.validateTranslations (translatedProject)
17     if validationReport.hasUncoveredFunctions () then
18         agentGeneratedTestCode ← validatorAgent.generateAndValidateTests ()
19         translatedProject ← translatedProject ∪ agentGeneratedTestCode
20     translatedValidatedProject ← translatedProject
21     return translatedValidatedProject, validationReport
22 !timeout : analyzerOutput ← runAnalyzerAgent (sourceProject)
23 !timeout : planningOutput ← runPlanningAgent (sourceProject, analyzerOutput)
24 context ← sourceProject ∪ analyzerOutput ∪ planningOutput
25 for iteration ← 1 to maxIter do
26     !timeout : translatedProject ← runTranslatorAgent (context, validationReport)
27     !timeout : translatedValidatedProject, validationReport ← runValidatorAgent
         (context, translatedProject)
28     if validationReport.isAllSuccess () then
29         break
30 return translatedValidatedProject
```

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`.

3.3.2 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.

3.3.3 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.

3.3.4 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* (§3.4) and *Validator* (§3.5) *Agents* to execute.

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 1, lines 1–13). 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*.

3.5 Validator Agent

The *Validator Agent* validates the functional correctness of the translated project (Algorithm 1, lines 14–21). 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.

3.5.1 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.

3.5.2 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.

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?

4.1 Experimental Setup

4.1.1 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 1 provides an overview of our open-source subject translation projects from four recent repository-level code translation techniques [18, 30, 51, 80] covering the following PLs: C, Go, Rust, Java, Python, and JavaScript. The current implementation of ReCodeAgent supports tools discussed in §3.1 for these six PLs. More languages can be supported by implementing `get_file_structure` with roughly 300–350 LoC per PL. The effort usually involves creating the parse tree and traversing it to extract specific code constructs, e.g., functions, classes, variables, etc. LSP tools do not require manual effort, it can be simply integrated by downloading the PL's language server protocol implementation. In total, our evaluation includes 118 projects spanning over 230K lines of code. We exclude Syzygy [51] from our evaluation due to the unavailability of its artifact. Since the test suites of RepoTransBench [68] are written by LLMs and not validated as correct by humans, we exclude them as well. For AlphaTrans [18], 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.

4.1.2 LLM. ReCodeAgent works with different LLMs. Major software engineering leaderboards [4] 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[3]. 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.

4.1.3 Competing Techniques. We compare ReCodeAgent against Skel [65], Oxidizer [80], AlphaTrans [18], and SWE-agent [73] from Crust [30] after reproducing their results with the same LLM. While we cannot directly compare to ACToR [31] 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[4].

4.1.4 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 [54] as our agent SDK for the agentic workflow discussed in §3. Specifically, ReCodeAgent uses Claude Code to run the main agent loop for each of the four agents discussed in §3. Moreover, we also use Claude Code's basic tools (e.g., `Read`, `Write`, etc.), however, ReCodeAgent implements the MCP tools discussed in §3.1.

[3]Please see our artifacts [48] on how to run with other models from OpenRouter.
[4]Confirmed by the authors of ACToR [31].

Table 1: 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-{$\alpha, \beta, \sigma, \gamma$}: α: both compile, β: only ReCodeAgent compile, σ: only SWE-agent compile, γ: both do not compile, C: Test coverage, C+: Test coverage with generated tests. 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 (Go→Rust)	checkdigit	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	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	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	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	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	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 (Java→Python)	cli	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	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	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	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 (Python→JavaScript)	bst	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	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	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	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	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	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	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	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 (C→Rust)	Crust-α	22961	⟨40, 40⟩	166	⟨153, 166⟩	⟨130, 146⟩	⟨23, 20⟩	-	-	-	493	493	0	68.2	75.7	673	-	-
	Crust-β	66704	⟨0, 49⟩	321	⟨0, 320⟩	⟨0, 295⟩	⟨0, 25⟩	-	-	-	1118	1114	4	57.9	75.2	1900	-	-
	Crust-σ	3894	⟨1, 0⟩	1	⟨1, 0⟩	⟨1, 0⟩	⟨0, 0⟩	-	-	-	53	51	2	4.3	67.4	41	-	-
	Crust-γ	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⟩

4.2 RQ1: Effectiveness of ReCodeAgent

Table 1 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* (§4.2.1), (2) *Test Validation* (§4.2.2), and (3) *Function Validation* (§4.2.3).

4.2.1 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`.

```
1 ---------- C SOURCE CODE ----------
2 int        writechar(char c, int *len)
  ↪  {
3     return ((*len)++, write(1, &c, 1));
4 }
```

```
1 ---------- RUST TRANSLATION ----------
2 pub fn writechar(c: char, len: &mut
  ↪  i32) -> i32 {
3     *len += 1; ...
4 }
```

4.2.2 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. We only used validated test suites after the execution of ReCodeAgent to ensure they remain hidden during experiments. Because Oxidizer does not translate tests, we manually translated and validated the Go tests into Rust. Multi-column *Validated Developer Tests* in Table 1 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-α). 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 SOURCE CODE ----------
public class CommandLine implements
    Serializable {
    protected CommandLine() {}
}
```

```
---------- 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 §4.3. Multi-column *ReCodeAgent Translated Developer Tests* in Table 1 summarize these results. Across 1,484 translated tests, excluding Crust where test translation is not required as their authors provide validated tests in Rust and include it in their prompts, 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 (§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.

Moreover, we manually investigated the quality and effectiveness of generated tests by ReCodeAgent. To perform the study, we created a benchmark of 502 tests uniformly sampled from all four existing techniques. Two human investigators independently analyzed each generated test and the method ReCodeAgent used to determine coverage gap with an inter-rater agreement of 100%. On average, our manual investigation show that ReCodeAgent can correctly generate an effective test which exercises previously uncovered code 71.1% of the time. To determine coverage gap, our analysis of agent trajectories indicate that for larger projects like AlphaTrans, ReCodeAgent generates internal scripts to extract uncovered code. For smaller projects in other benchmarks, it simply relies on reading source and test code. The details of our human study with comments from investigators is publicly available [48].

4.2.3 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 [18] 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);
    ...
}
```

4.3 RQ2: Test Translation

Table 2 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 structurally similar, 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

Table 2: Comparison between ReCodeAgent translated tests and original source PL tests. Tuple entries indicate ⟨Source Test, Translated Test⟩. LoC: Lines of Code.

Tool	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⟩

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` [58]. 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.

Beside non-functional metrics discussed above, we also performed manual investigation to assess the quality of translated tests. To evaluate tests across multiple benchmarks, we sampled 100 translated tests from AlphaTrans and Oxidizer, and used all 74 tests from Skel. To do the human study, we first leveraged two state-of-the-art LLMs, namely, `claude-sonnet-5` and `claude-opus-4-8` to perform initial analysis on the quality of the translated tests. Two human investigators then manually checked their analysis to ensure correctness and met with each other to resolve conflicts. The results from our human study of translated tests indicate ReCodeAgent achieves 88% accuracy with an inter-rater agreement of 96.7%. Please refer to our artifacts for more comprehensive comments by each manual investigator [48].

4.4 RQ3: Ablation Study

Figure 7 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-α), and Base Agent with Prompt Concatenation (BA-β). The top portion of Figure 7 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.

4.4.1 Impact of Individual Agents. Removing the analyzer agent (NoA) results in decreased test validation performance (↓22.7%) across all benchmarks. Without foundational analysis of the source

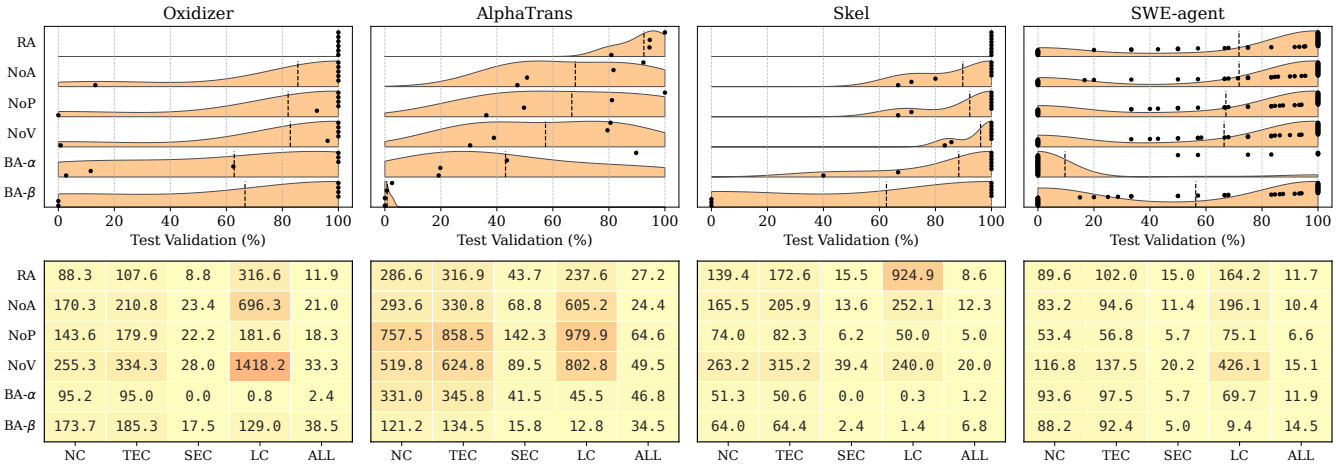


Figure 7: Impact of different agents on translation effectiveness and agent trajectories. RA: ReCodeAgent, NoA: No Analyzer, NoP: No Planning, NoV: No Validator, BA-α: Base Agent with Prompt Condensation, BA-β: Base Agent with Prompt Concatenation, NC: Node Count, TEC: Temporal Edge Count, SEC: Structural Edge Count, LC: Loop Count, ALL: Average Loop Length.

project structure and third-party library 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.

4.4.2 Comparison with Base Agents. The base agent configurations represent ReCodeAgent without specialized agents. BA-α condenses the entire translation task into a single compact prompt, while BA-β 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-α and 62.4% for BA-β. 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.

4.4.3 Trajectory Analysis. To perform a process-centric analysis of agent trajectories, we use Graphectory [34] metrics. These metrics carry the same interpretation in code translation similar to SWE-bench for fixing software engineering bugs, e.g., a higher node count (NC), temporal edge count (TEC), or loop count (LC) indicates the agent takes more actions to solve the problem. Our objective is to show that test validation degradation alone is insufficient for evaluating ablations. The heatmaps in Figure 7 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 NC and TEC. For example, NoV removes the validator agent and iterative feedback, depriving the translator

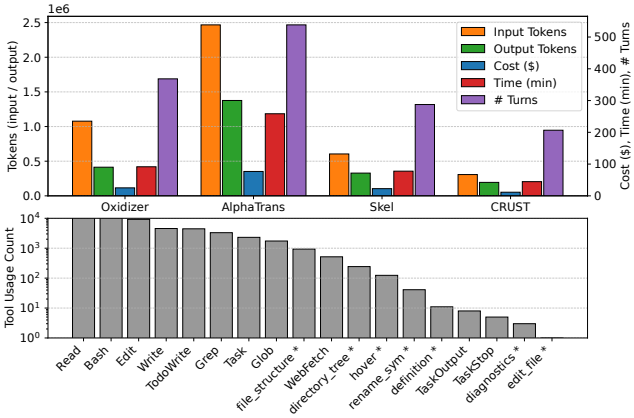


Figure 8: Cost and Tool Usage Analysis of ReCodeAgent.

agent of independent quality assurance. The NoV ablation shows higher NC, TEC, and LC because the translator agent performs both translation and validation within the same context window. This increased workload often leads to memory issues and higher LC. ReCodeAgent avoids this by using the validator to simplify the translator's task. These process-centric metrics consistently show that ReCodeAgent's multi-agent architecture provides structured guidance that prevents unnecessary exploration and repeated work.

4.5 RQ4: Cost and Tool Usage Analysis

Figure 8 presents ReCodeAgent's computational costs per project and tool utilization patterns across all four benchmarks.

4.5.1 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

Table 3: Cost comparison of existing techniques and ReCodeAgent averaged by number of projects translated.

Tool	Input Tokens	Output Tokens	Time (s)	Cost ($)
Oxidizer	131085.0	44855.2	940.4	1.1
AlphaTrans	2302094.0	296475.2	37503.4	11.4
Skel	45763.5	10733.6	450.8	0.3
Crust	1060519.4	11899.7	455.1	4.0
ReCodeAgent	446829.7	257879.0	3433.2	15.3

smaller individual project sizes. This predictable scaling enables users to estimate costs for new projects based on codebase size.

Moreover, Table 3 shows the cost comparison between existing techniques and ReCodeAgent averaged over number of translated projects. On average, it costs $15.3 and 57 minutes for ReCodeAgent to translate and validate each project. The cost difference between ReCodeAgent and existing techniques is justifiable as existing techniques can only translate *a single PL pair* and have high development cost to generalize across other PLs.

4.5.2 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.

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 [20], CxGo [63], Sharpen [46], and Java2CSharp [21] translate code from C to Rust, C to Go, and Java to C#, respectively. A series of statistical machine translation techniques [7, 39–41] focus on translating Java to C#. Deep learning approaches have also been applied for code translation [49, 50]. Recent advancements have focused on using LLMs for code translation [12, 25, 45, 62, 71, 78], 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 [18] is a neuro-symbolic technique with a deterministic pipeline—which includes program decomposition, type resolution, skeleton generation, etc. and is not autonomous like agentic approaches, e.g., ReCodeAgent—to enable repository-level code translation and translates Java to Python using open-source LLMs and GraalVM [44] for isolated validation; Syzygy [51] targets C to Rust using GPT-4; Skel [65] is also a neuro-symbolic technique that decomposes source project into smaller pieces, generates skeletons in target language and uses program execution traces for validation and translates Python to JavaScript; Oxidizer [80] is another neuro-symbolic approach that employs type-driven techniques and language feature mapping to convert Go to Rust. Oxidizer extracts input/output pairs by executing source project tests and generates mock tests in target language for validating translations. Some approaches have combined transpiler outputs with LLM-based translation [72], but their success is often limited by the availability and reliability of the underlying transpilers. Nitin et al. [42] capture natural language specifications from source code to inform translation, while Yang et al. [76] utilize test cases to support the process.

The rise of agent-based frameworks [8, 9, 33, 35, 69] has produced significant research and industrial interest in applying these architectures to a variety of software engineering challenges [10, 26, 75]. SWE-agent [74] introduces a specialized agent-computer interface (ACI), enabling agents to interact with code repositories through file reading, editing, and execution of bash commands. SWE-agent from Crust [30] and MatchFixAgent [19] are the only autonomous agentic approach that has been adapted for code translation. Similar to ReCodeAgent, they follows a cycle of reasoning and action to either only translate code, or only validate and repair the code translations. However, ReCodeAgent automates the entire translation and validation pipeline into a multi-agent scaffold, improving the overall performance compared to these related approaches.

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 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.

8 Data Availability Statement

The artifacts of ReCodeAgent are publicly available [48].

Acknowledgments

This work is supported by NSF CCF-2238045 grant. We thank the anonymous reviewers for their comments, which helped make this work stronger. We also thank Professor Elsa Gunter for her support throughout this project.

References

[1] Muhammad Salman Abid, Mrigank Pawagi, Sugam Adhikari, Xuyan Cheng, Ryed Badr, Md Wahiduzzaman, Vedant Rathi, Ronghui Qi, Choiyin Li, Lu Liu, Rohit Sai Naidu, Licheng Lin, Que Liu, Asif Zubayer Palak, Mehzabin Haque, Xinyu Chen, Darko Marinov, and Saikat Dutta. 2024. GlueTest: Testing Code Translation via Language Interoperability. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 612–617. doi:10.1109/ICSME58944.2024.00061

[2] Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu Lahiri, and Sriram Rajamani. 2023. Monitor-guided decoding of code lms with static analysis of repository context. In *Advances in Neural Information Processing Systems*, Vol. 36. 32270–32298. https://neurips.cc/media/neurips-2023/Slides/70362.pdf

[3] Toufique Ahmed, Jatin Ganhotra, Avraham Shinnar, and Martin Hirzel. 2026. Investigating Test Overfitting on SWE-bench. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering* (Concordia University, Montreal, QC, Canada) *(FSE Companion '26)*. Association for Computing Machinery, New York, NY, USA, 1292–1296. doi:10.1145/3803437.3805574

[4] The SWE bench Team. 2026. SWE-bench Leaderboard. https://www.swebench.com/

[5] Xuemeng Cai, Jiakun Liu, Xiping Huang, Yijun Yu, Haitao Wu, Chunmiao Li, Bo Wang, Imam Nur Bani Yusuf, and Lingxiao Jiang. 2025. RustMap: Towards Project-Scale C-to-Rust Migration via Program Analysis and LLM. In *Engineering of Complex Computer Systems: 29th International Conference, ICECCS 2025, Hangzhou, China, July 2–4, 2025, Proceedings* (Hangzhou, China). Springer-Verlag, Berlin, Heidelberg, 283–302. doi:10.1007/978-3-032-00828-2_16

[6] Kevin Chen, Marco Cusumano-Towner, Brody Huval, Aleksei Petrenko, Jackson Hamburger, Vladlen Koltun, and Philipp Krähenbühl. 2025. Reinforcement learning for long-horizon interactive llm agents. *arXiv preprint arXiv:2502.01600* (2025).

[7] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems* 31 (2018).

[8] Yang Chen, Aliya Ahmad, Yiheng Zhou, and Reyhaneh Jabbarvand. 2026. Unlocking Model Potentials Through Adaptive Multi-Agent Scaffolding for Efficient Issue Resolution. *arXiv preprint arXiv:2606.25514* (2026).

[9] Yang Chen, Toufique Ahmed, Reyhaneh Jabbarvand, and Martin Hirzel. 2026. Can Old Tests Do New Tricks for Resolving SWE Issues? *Proc. ACM Softw. Eng.* 3, FSE, Article FSE141 (June 2026), 22 pages. doi:10.1145/3808148

[10] Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/

[11] Saman Dehghan, Tianran Sun, Tianxiang Wu, Zihan Li, and Reyhaneh Jabbarvand. 2025. Translating Large-Scale C Repositories to Idiomatic Rust. *arXiv preprint arXiv:2511.20617* (2025).

[12] Peng Di, Jianguo Li, Hang Yu, Wei Jiang, Wenting Cai, Yang Cao, Chaoyu Chen, Dajun Chen, Hongwei Chen, Liang Chen, Gang Fan, Jie Gong, Zi Gong, Wen Hu, Tingting Guo, Zhichao Lei, Ting Li, Zheng Li, Ming Liang, Cong Liao, Bingchang Liu, Jiachen Liu, Zhiwei Liu, Shaojun Lu, Min Shen, Guangpei Wang, Huan Wang, Zhi Wang, Zhaogui Xu, Jiawei Yang, Qing Ye, Gehao Zhang, Yu Zhang, Zelin Zhao, Xunjin Zheng, Hailian Zhou, Lifu Zhu, and Xianying Zhu. 2024. CodeFuse-13B: A Pretrained Multi-lingual Code Large Language Model. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice* (Lisbon, Portugal) *(ICSE-SEIP '24)*. Association for Computing Machinery, New York, NY, USA, 418–429. doi:10.1145/3639477.3639719

[13] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-Collaboration Code Generation via ChatGPT. *ACM Trans. Softw. Eng. Methodol.* 33, 7, Article 189 (Sept. 2024), 38 pages. doi:10.1145/3672459

[14] Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2025. PLAN-AND-ACT: improving planning of agents for long-horizon tasks. In *Proceedings of the 42nd International Conference on Machine Learning* (Vancouver, Canada) *(ICML'25)*. JMLR.org, Article 594, 44 pages.

[15] GitHub. 2026. CodeQL. https://codeql.github.com

[16] Ziqi Guan, Xin Yin, Zhiyuan Peng, and Chao Ni. 2025. Repotransagent: Multi-agent llm framework for repository-aware code translation. *arXiv preprint arXiv:2508.17720* (2025).

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

[18] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2025. AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE109 (June 2025), 23 pages. doi:10.1145/3729379

[19] Ali Reza Ibrahimzada, Brandon Paulsen, Reyhaneh Jabbarvand, Joey Dodds, and Daniel Kroening. 2026. MatchFixAgent: Language-Agnostic Autonomous Repository-Level Code Translation Validation and Repair. In *Forty-third International Conference on Machine Learning*. https://openreview.net/forum?id=MuyXpH3GL1

[20] Immunant. 2024. C2Rust Transpiler. https://github.com/immunant/c2rust

[21] Paul Irwin. 2026. Java to CSharp Converter. https://github.com/paulirwin/JavaToCSharp

[22] Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. 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)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 4912–4944. doi:10.18653/v1/2024.acl-long.269

[23] Suman Jain and Inderveer Chana. 2015. Modernization of Legacy Systems: A Generalised Roadmap. In *Proceedings of the Sixth International Conference on Computer and Communication Technology 2015* (Allahabad, India) *(ICCCT '15)*. Association for Computing Machinery, New York, NY, USA, 62–67. doi:10.1145/2818567.2818579

[24] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. 2013. Cloud Migration Research: A Systematic Review. *IEEE Transactions on Cloud Computing* 1, 2 (2013), 142–157. doi:10.1109/TCC.2013.10

[25] Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. 2024. On the Evaluation of Neural Code Translation: Taxonomy and Benchmark. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Echternach, Luxembourg) *(ASE '23)*. IEEE Press, 1529–1541. doi:10.1109/ASE56229.2023.00114

[26] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[27] Kaiyao Ke, Ali Reza Ibrahimzada, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2025. Advancing Automated In-Isolation Validation in Repository-Level Code Translation. *arXiv preprint arXiv:2511.21878* (2025).

[28] Ravi Khadka, Belfrit V. Batlajery, Amir M. Saeidi, Slinger Jansen, and Jurriaan Hage. 2014. How do professionals perceive legacy systems and software modernization?. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 36–47. doi:10.1145/2568225.2568318

[29] Musawwer Khan, Islam Ali, Wasif Nisar, Muhammad Qaiser Saleem, Ali S. Ahmed, Haysam E. Elamin, Waqar Mehmood, and Muhammad Shafiq. 2022. Modernization Framework to Enhance the Security of Legacy Information Systems. *Intelligent Automation & Soft Computing* 32, 1 (2022), 543–555. doi:10.32604/iasc.2022.016120

[30] Anirudh Khatry, Robert Zhang, Jia Pan, Ziteng Wang, Qiaochu Chen, Greg Durrett, and Isil Dillig. 2025. CRUST-Bench: A Comprehensive Benchmark for C-to-safe-Rust Transpilation. In *Second Conference on Language Modeling*. https://openreview.net/forum?id=8xofWL61S9

[31] Tianyu Li, Ruishi Li, Bo Wang, Brandon Paulsen, Umang Mathur, and Prateek Saxena. 2025. Adversarial Agent Collaboration for C to Rust Translation. *arXiv preprint arXiv:2510.03879* (2025).

[32] Zi Lin, Sheng Shen, Jingbo Shang, Jason E Weston, and Yixin Nie. 2025. 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*. https://openreview.net/forum?id=j6tMZaPWWF

[33] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2026. Large Language Model-Based Agents for Software Engineering: A Survey. *ACM Trans. Softw. Eng. Methodol.* (March 2026). Just Accepted. doi:10.1145/3796507

[34] Shuyang Liu, Yang Chen, Rahul Krishna, Saurabh Sinha, Jatin Ganhotra, and Reyhaneh Jabbarvand. 2026. Process-Centric Analysis of Agentic Software Systems. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 163 (April 2026), 28 pages. doi:10.1145/3798271

[35] Shuyang Liu, Saman Dehghan, Jatin Ganhotra, Martin Hirzel, and Reyhaneh Jabbarvand. 2026. From Plan to Action: How Well Do Agents Follow the Plan? *arXiv preprint arXiv:2604.12147* (2026).

[36] Feng Luo, Kexing Ji, Cuiyun Gao, Shuzheng Gao, Jia Feng, Kui Liu, Xin Xia, and Michael R. Lyu. 2025. Integrating Rules and Semantics for LLM-Based C-to-Rust Translation. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 685–696. doi:10.1109/ICSME64153.2025.00069

[37] Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. 2024. LLM Critics Help Catch LLM Bugs. *arXiv preprint arXiv:2407.00215* (2024).

[38] Microsoft. 2026. Language Server Implementations. https://microsoft.github.io/language-server-protocol/implementors/servers/

[39] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia)

(ESEC/FSE 2013). Association for Computing Machinery, New York, NY, USA, 651–654. doi:10.1145/2491411.2494584

[40] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2014. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 544–547. doi:10.1145/2591062.2591072

[41] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2015. 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* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 585–596. doi:10.1109/ASE.2015.74

[42] Vikram Nitin, Rahul Krishna, and Baishakhi Ray. 2024. Spectra: Enhancing the code translation ability of language models by generating multi-modal specifications. *arXiv preprint arXiv:2405.18574* (2024).

[43] Vikram Nitin, Rahul Krishna, Luiz Lemos do Valle, and Baishakhi Ray. 2026. C2SaferRust: Transforming C Projects Into Safer Rust With NeuroSymbolic Techniques. *IEEE Transactions on Software Engineering* 52, 2 (2026), 618–630. doi:10.1109/TSE.2025.3641486

[44] Oracle. 2026. GraalVM. https://www.graalvm.org.

[45] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 82, 13 pages. doi:10.1145/3597503.3639226

[46] Mono Project. 2026. Sharpen - Automated Java->C# coversion. https://github.com/mono/sharpen

[47] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative Agents for Software Development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 15174–15186. doi:10.18653/v1/2024.acl-long.810

[48] ReCodeAgent. 2026. Artifact Website. https://doi.org/10.5281/zenodo.21842351.

[49] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in neural information processing systems* 33 (2020), 20601–20611.

[50] Baptiste Roziere, Jie Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging Automated Unit Tests for Unsupervised Code Translation. In *International Conference on Learning Representations*. https://openreview.net/forum?id=cmt-6KtR4c4

[51] Manish Shetty, Naman Jain, Adwait Godbole, Sanjit A Seshia, and Koushik Sen. 2024. Syzygy: Dual Code-Test C to (safe) Rust Translation using LLMs and Dynamic Analysis. *arXiv preprint arXiv:2412.14234* (2024).

[52] HoHyun Sim, Hyeonjoong Cho, Yeonghyeon Go, Zhoulai Fu, Ali Shokri, and Binoy Ravindran. 2025. Large Language Model-Powered Agent for C to Rust Code Translation. *arXiv preprint arXiv:2505.15858* (2025).

[53] Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. 2026. Scaling Long-Horizon Agent via Context Folding. In *Forty-third International Conference on Machine Learning*. https://openreview.net/forum?id=lNRgWoGfYg

[54] The Claude Code Team. 2026. Claude Code. https://github.com/anthropics/claude-code

[55] The Eclipse Team. 2026. Eclipse JDT Language Server. https://github.com/eclipse-jdtls/eclipse.jdt.ls

[56] The Go Team. 2026. Gopls: The language server for Go. https://go.dev/gopls/

[57] The LLVM Team. 2026. clangd. https://github.com/clangd/clangd

[58] The Qwen Team. 2026. Qwen Embedding. https://huggingface.co/Qwen/Qwen3-Embedding-0.6B

[59] The Rust Language Team. 2026. Rust Analyzer. https://rust-analyzer.github.io/

[60] The Spyder IDE Team. 2026. Python LSP Server. https://github.com/python-lsp/python-lsp-server

[61] The TypeScript Language Server Team. 2026. TypeScript Language Server. https://github.com/typescript-language-server/typescript-language-server

[62] Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. 2024. StructCoder: Structure-Aware Transformer for Code Generation. *ACM Trans. Knowl. Discov. Data* 18, 3, Article 70 (Jan. 2024), 20 pages. doi:10.1145/3636430

[63] Go Transpile. 2024. C to Go Translator. https://github.com/gotranspile/cxgo

[64] Tree-Sitter. 2026. Tree-Sitter Library. https://tree-sitter.github.io/tree-sitter/

[65] Bo Wang, Tianyu Li, Ruishi Li, Umang Mathur, and Prateek Saxena. 2025. Program Skeletons for Automated Program Translation. *Proc. ACM Program. Lang.* 9, PLDI, Article 184 (June 2025), 25 pages. doi:10.1145/3729287

[66] Chaofan Wang, Tingrui Yu, Beijun Shen, Jie Wang, Dong Chen, Wenrui Zhang, Yuling Shi, Chen Xie, and Xiaodong Gu. 2026. EvoC2Rust: A Skeleton-guided Framework for Project-Level C-to-Rust Translation. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '26)*. Association for Computing Machinery, New York, NY, USA, 94–105. doi:10.1145/3786583.3786856

[67] Yanlin Wang, Rongyi Ou, Yanli Wang, Mingwei Liu, Jiachi Chen, Ensheng Shi, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2026. EffiReasonTrans: RL-Optimized Reasoning for Code Translation. *IEEE Transactions on Software Engineering* (2026), 1–12. doi:10.1109/TSE.2026.3693795

[68] Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. 2026. RepoTransBench: A Real-World Multilingual Benchmark for Repository-Level Code Translation . *IEEE Transactions on Software Engineering* 52, 02 (Feb. 2026), 675–690. doi:10.1109/TSE.2025.3645056

[69] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, Qi Zhang, and Tao Gui. 2025. The rise and potential of large language model based agents: a survey. *Science China Information Sciences* 68, 2 (17 Jan 2025), 121101. doi:10.1007/s11432-024-4222-0

[70] Pengyu Xue, Linhao Wu, Zhen Yang, Chengyi Wang, Xiang Li, Yuxiang Zhang, Jia Li, Ruikai Jin, Yifei Pei, Zhaoyan Shen, Xiran Lyu, and Jacky Wai Keung. 2025. ClassEval-T: Evaluating Large Language Models in Class-Level Code Translation. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA063 (June 2025), 24 pages. doi:10.1145/3728940

[71] Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. CodeTransOcean: A Comprehensive Multilingual Benchmark for Code Translation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 5067–5089. doi:10.18653/v1/2023.findings-emnlp.337

[72] Aidan Z.H. Yang, Yoshiki Takashima, Brandon Paulsen, Josiah Dodds, and Daniel Kroening. 2025. 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, 1453–1463. doi:10.1109/ASE63991.2025.00123

[73] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: agent-computer interfaces enable automated software engineering. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 1601, 125 pages.

[74] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=mXpq6ut8J3

[75] John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, Diyi Yang, Sida Wang, and Ofir Press. 2025. SWE-bench Multimodal: Do AI Systems Generalize to Visual Software Domains?. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=riTiq3i21b

[76] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and Unleashing the Power of Large Language Models in Automated Code Translation. *Proc. ACM Softw. Eng.* 1, FSE, Article 71 (July 2024), 24 pages. doi:10.1145/3660778

[77] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

[78] Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. 2024. Rectifier: Code translation with corrector via llms. *arXiv preprint arXiv:2407.07472* (2024).

[79] Zhiqiang Yuan, Wenjun Mao, Zhuo Chen, Xiyue Shang, Chong Wang, Yiling Lou, and Xin Peng. 2026. Project-Level C-to-Rust Translation via Pointer Knowledge Graphs. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE162 (June 2026), 24 pages. doi:10.1145/3808169

[80] Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. 2025. Scalable, Validated Code Translation of Entire Projects using Large Language Models. *Proc. ACM Program. Lang.* 9, PLDI, Article 212 (June 2025), 26 pages. doi:10.1145/3729315

[81] Tianyang Zhou, Ziyi Zhang, Haowen Lin, Somesh Jha, Mihai Christodorescu, Kirill Levchenko, and Varun Chandrasekaran. 2026. SACTOR: LLM-Driven Correct and Idiomatic C to Rust Translation with Static Analysis and FFI-Based Verification. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (Eds.). Association for Computational Linguistics, San Diego, California, United States, 646–673. doi:10.18653/v1/2026.acl-long.28

Received 2026-03-26; accepted 2026-06-18