AnchorRepair: A Reference-Guided Dual-Agent Framework for Code Translation Repair

Jiajun Sun
School of Computer Software, Tianjin University
Tianjin, China
sjjtianjin@tju.edu.cn

Yaocai Zhao
School of Computer Software, Tianjin University
Tianjin, China
zyc456@tju.edu.cn

Fengjie Li
School of Computer Software, Tianjin University
Tianjin, China
fengjie@tju.edu.cn

Jiajun Jiang*
School of Computer Software, Tianjin University
Tianjin, China
jiangjiajun@tju.edu.cn

Abstract

Automated code translators can now produce repository-scale translations from Java to Python, yet a substantial fraction of translated functions remain semantically incorrect. Repairing such defects differs from traditional automated program repair (APR) because the correctness specification is based on the source language and no target-language test suite exists to guide fault localization. We observe that the original Java code is the most authoritative specification for repair and propose AnchorRepair, a dual-agent framework that treats the Java source as first-class diagnostic evidence. The Diagnosis Agent operates in a ReAct-style loop and autonomously invokes seven cross-lingual analysis tools, producing a structured root-cause diagnosis by contrasting the Java specification against the Python translation across control flow, data flow, and call graph dimensions. A Patch Agent then generates a corrected Python function conditioned on this diagnosis and the retrieved Java context. We evaluated AnchorRepair on 82 real translation defects from four Apache Commons projects. With DeepSeek-V3.2 as the backend, AnchorRepair repairs 40 defects (48.8%), a 12-point absolute improvement over the LLM-Only baseline (28/82) and substantially outperforming TransAgent (17/82) and D4C (21/82). The framework generalizes across LLM backends without modification and achieves repair coverage comparable to a general-purpose coding agent at less than half the cost. Our source code and all experimental results are publicly available at https://github.com/JJS-TJ/AnchorRepair.

CCS Concepts

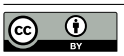
• **Software and its engineering** → **Software maintenance tools**.

Keywords

Code Translation, Automated Program Repair, LLM Agent

ACM Reference Format:
Jiajun Sun, Yaocai Zhao, Fengjie Li, and Jiajun Jiang. 2026. AnchorRepair: A Reference-Guided Dual-Agent Framework for Code Translation Repair. In *Proceedings of the 7th International Workshop on Automated Program Repair (APR '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3843776.3844669

*Jiajun Jiang is the corresponding author.



APR '26, Munich, Germany

ACM ISBN 979-8-4007-2988-1/2026/10
https://doi.org/10.1145/3843776.3844669

1 Introduction

Code translation converts a program from one programming language to another while preserving its functionality [26]. It has broad applications, including migrating legacy systems in outdated languages [23], transitioning to faster languages for performance-critical scenarios [29], and supporting cross-language interoperability in multi-language organizations [24]. Automated code translation reduces manual effort and has attracted widespread attention in recent years [10, 15, 36, 45].

Recent advances in large language models (LLMs) have improved code translation, yielding promising results on standardized benchmarks [25, 40]. Based on translation granularity, existing work can be classified into three levels [26]. **Snippet-level** translation targets isolated code fragments between consecutive comments [46, 47]. **Function-level** translation handles complete functions with type signatures and local dependencies [1, 13, 22]. **Repository-level** translation aims to convert entire projects with hundreds of interdependent files, class hierarchies, and build configurations [11, 30]. This level is the most challenging. Pan et al. [26] show that current LLMs struggle to translate entire repositories correctly. Tools like AlphaTrans [11] still produce numerous semantic defects despite high syntactic validity.

These translation-induced defects differ qualitatively from traditional single-language bugs. The generated code is syntactically valid and locally plausible, yet semantically misaligned with the source implementation. Diagnosing such misalignment requires reasoning about both languages, since fault evidence resides in the source language rather than the target.

A key observation motivates our approach. **The original Java code is correct and constitutes the most authoritative specification of intended behavior.** As shown in Figure 1, the translated code is syntactically valid and locally plausible, yet semantically wrong. To correct this defect, a repair tool must consult the Java source to understand that the field uses `null` rather than empty string as its sentinel value. This type of cross-lingual evidence is unavailable to traditional automated program repair (APR) techniques, which operate exclusively within the target language.

```
public static Option create(final String opt) {        Correct Java Code (specification)
        Option option = null;
        try {
             option = Option. Option1(opt, description);  ...
            }
        return option
```

```
def create (opt: str)                                  Buggy Python Code (translation)
+    option = None
     try:
        option=Option.Option1(opt, OptionBuilder.__description) ...
     return option
```

Figure 1: An example of translated buggy function.

Automated program repair (APR) has been extensively studied for single-language bugs, with LLM-based approaches achieving notable success on benchmarks such as Defects4J [18]. Translation repair is different because traditional APR relies on a same-language test suite to define correctness and guide fault localization, whereas in translation repair the specification is encoded in the source language and no target-language tests exist.

These differences violate two critical assumptions that underpin most APR methods. **First**, many APR techniques [14, 20, 37, 39, 43] rely on per-method test cases to guide fault localization or execution-aligned debugging. In the code translation setting, method-level test suites in the target language are unavailable. Translating the test suite itself is not a viable alternative, because a translation error in one method can propagate through test execution and mask or misattribute failures in other methods. For example, if a test invokes four methods but one of them is mistranslated, the resulting runtime error prevents validation of the remaining three. GraalVM [31] overcomes this coupling effect by executing the original Java tests directly against individual translated Python methods via polyglot interoperability. Each method is validated in isolation using the source-language tests directly, without translation. However, GraalVM provides only pass/fail verdicts and does not offer the fine-grained execution traces that APR tools rely on. **Second**, existing approaches treat each function as an independent repair unit and overlook the inter-function coupling that characterizes real-world projects. Without class-level context from the Java source, isolated repair may inadvertently break cross-method contracts.

To bridge these gaps, we propose AnchorRepair, a **dual-agent repair framework** that treats the Java source as diagnostic evidence. The key insight is that translation defects manifest as semantic divergences between the two implementations, which can be systematically exposed through differential analysis. The **Diagnosis Agent** is grounded in the ReAct [41] paradigm and autonomously orchestrates seven analysis tools that span two complementary families. **Semantic and structural differential analyzers** detect mismatches in control flow, data flow, and call graphs between the Java and Python implementations. **Source-context retrievers** grant the agent direct visibility into Java source files, functions, and their JavaDoc. These tools replace the missing test-driven feedback with direct, multi-dimensional access to Java program semantics. Based on the structured root-cause diagnosis, the *Patch Agent* generates a corrected Python function informed by both the identified defect and the relevant Java evidence.

We evaluate AnchorRepair on 82 real-world Java-to-Python translation defects spanning four Apache Commons projects. With DeepSeek-V3.2 as the backbone LLM, AnchorRepair repairs 40 out of 82 defects (48.8%), a **12-point absolute improvement** over the LLM-Only baseline (28/82). It substantially outperforms TransAgent (17/82) and D4C (21/82). Experiments with Qwen3.5-Plus and Claude-Sonnet-4.6 confirm that the framework generalizes across model families without modification. A comparison with Claude Code demonstrates that our domain-specific toolkit achieves comparable repair coverage at less than half the cost.

In summary, this paper makes the following contributions.

- We identify that the original Java source is the most valuable yet underutilized evidence for repairing translation-induced defects. We propose a dual-agent framework that systematically exploits this cross-lingual signal through seven tools.
- We design a dual-agent repair framework that decouples diagnosis from patch generation. The Diagnosis Agent leverages seven static analysis tools to systematically compare Java and Python code across multiple dimensions and replaces test-driven feedback with direct semantic analysis.
- We evaluate AnchorRepair on 82 real-world defects and demonstrate substantial improvements over both the LLM-Only baseline and two state-of-the-art translation repair methods under controlled conditions.
- We provide analyses of tool usage patterns, LLM backend generalization, and cost-effectiveness trade-offs. These analyses offer practical guidance for deploying translation repair.
- We release our implementation, benchmark, and all experimental results for replication and future research. https://github.com/JJS-TJ/AnchorRepair

2 Related Work

2.1 Code Translation

Code translation techniques have evolved from rule-based transpilers to learning-based and LLM-driven approaches. Traditional transpiler tools such as C2Rust [2] and JavaToCSharp [3] rely on manually crafted transformation rules, which require substantial human effort and often produce code with poor readability. Deep learning approaches such as TransCoder [19] train models on monolingual corpora to learn unsupervised translation patterns, while pre-trained models like CodeBERT [9] and UniXcoder [10] provide general code representations that benefit downstream translation tasks.

Recent advances in LLMs have substantially improved translation quality [25, 26, 40]. However, as translation scales from isolated functions to entire repositories, semantic correctness degrades sharply. AlphaTrans [11] addresses repository-level translation through neuro-symbolic compositional decomposition with GraalVM-based validation. Pan et al. [26] show that LLM translators introduce a distinctive class of bugs that are syntactically plausible yet semantically wrong. Multi-agent translation systems [43] and benchmark studies [30] further characterize the difficulty of repository-level translation.

Prior work concentrates on producing translations and on detecting that they are faulty. MatchFixAgent [12] is the most similar to our work, but their method generates tests for the translated code to determine whether repair is needed. AnchorRepair is complementary: it starts where these systems leave off by taking a validated-as-incorrect translation and repairing it through reasoning across the language boundary not the test.

2.2 Automated Program Repair

Automated program repair (APR) has progressed from traditional [15, 16, 38] and learning-based methods [6, 17] to LLM-based approaches, including ChatRepair [34], ThinkRepair [42], and SRepair [35]. These methods leverage LLMs with stronger code understanding and generation capabilities and reformulate the APR task as one of effectively supplying the LLM with relevant knowledge and feedback to facilitate the repair process. Most recently, the literature on automated program repair has been dominated by agent-based and tool-augmented paradigms, such as AutoCodeRover [44], SpecRover [27], SWE-agent [39], and Agentless [33].

These systems are designed for faults whose intended behavior is specified only by failing tests, so their machinery centers on test-driven, iterative refinement. Translation repair differs fundamentally in its signal: the intended behavior is given explicitly by a correct program in the source language. AnchorRepair exploits this asymmetry by grounding diagnosis in cross-lingual differential analysis against the Java specification rather than inferring intent from test failures alone.

2.3 Tool-Augmented LLM Agents

The rise of agent-based frameworks [21, 32] has produced significant interest in applying LLM agents to software engineering tasks [33, 39, 44]. ReAct [41] interleaves natural-language reasoning with tool actions to improve grounding and interpretability. Toolformer [28] demonstrates that models can learn to invoke external tools autonomously. SpecRover [27] extends code-search agents with specification inference and generates function summaries and targeted feedback at key decision points.

AnchorRepair adopts the ReAct control loop but instantiates it with a purpose-built suite of cross-lingual analysis tools. These tools include semantic or structural differential analyzers and Java-side context retrievers tailored to diagnosing translation defects. AnchorRepair further pairs the diagnosing agent with a dedicated patching agent so that fault localization and repair are handled by separate, specialized roles. Whereas general agentic frameworks emphasize open-ended repository interaction, our contribution is the design of a focused instrument set whose observations map onto the concrete ways a translation can diverge from its specification.

3 Approach

We propose AnchorRepair, a tool-augmented dual-agent framework for repairing semantic defects introduced during Java-to-Python code translation. We first provide an overview of the framework and then describe each component in detail. These components include the Diagnosis Agent, the diagnostic tool suite, the Patch Agent, and the validation pipeline.

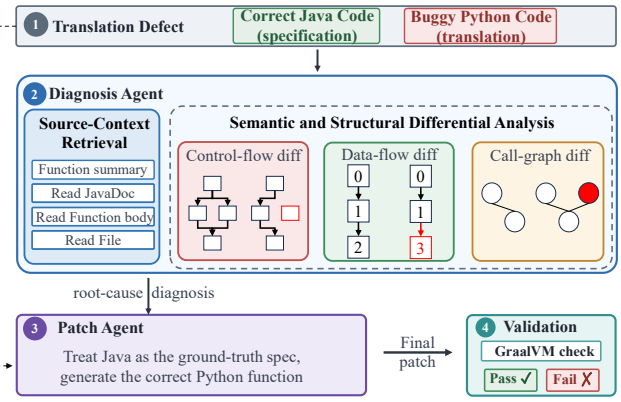


Figure 2: Overview of AnchorRepair.

3.1 Overview

The central premise of AnchorRepair is that the original Java code is correct and constitutes an authoritative behavioral specification. Unlike conventional program repair, where intended behavior must be inferred from test suites, a translation defect is accompanied by this specification explicitly. AnchorRepair exploits this asymmetry through a diagnose-then-patch architecture.

Figure 2 illustrates the overall pipeline. Given a translation defect (f_J, f_P), where f_J is the correct Java function and f_P is the buggy Python translation, AnchorRepair proceeds in three stages.

(1) **Diagnosis.** The Diagnosis Agent autonomously invokes a suite of seven analysis tools to gather cross-lingual evidence from f_J and f_P. It follows a ReAct-style [41] reasoning loop and interleaves natural-language thought with tool calls. It terminates by emitting a structured root-cause diagnosis d.

(2) **Patch generation.** The Patch Agent receives the diagnosis d, the Java specification f_J, and the buggy code f_P. It regenerates a corrected Python function $f_P^\star$ that preserves the original signature.

(3) **Validation.** The candidate patch $f_P^\star$ is validated by running the original Java test suite against the patched Python code via GraalVM cross-language execution.

Two design principles distinguish AnchorRepair from prior work. First, diagnosis and patching are **decoupled**. Rather than asking a single model to fix the code end-to-end, we make fault localization an explicit, tool-grounded step. This mirrors how developers debug translated code. Developers first understand **why** the ported code disagrees with the original and then edit. Second, the diagnostic tools treat the Java side as authoritative. Differential analyzers contrast f_J against f_P to identify discrepancies, and context retrievers surface additional Java evidence. The agent's reasoning is thus anchored to the correct specification rather than to the faulty translation.

3.2 Diagnosis Agent

The Diagnosis Agent is responsible for identifying the root cause of a translation defect. It operates in a ReAct-style [41] loop in which the agent reasons about the current evidence at each step

Table 1: The diagnostic tool suite.

Tool	Input	Evidence provided
`control_flow`	f_J, f_P	Control-flow mismatch (branches, loops, returns)
`data_flow`	f_J, f_P	Data-flow mismatch (variables, assignments, operators)
`call_graph`	f_J, f_P	Call-graph mismatch (missing or altered invocations)
`summarize_function`	f_J	LLM-generated summary of function
`read_function_doc`	f_J	JavaDoc comment
`read_function_code`	f_J	Full source code of a named Java function
`read_file_code`	f_J	Entire Java file

and autonomously decides whether to invoke one of the seven diagnostic tools or to terminate with a diagnosis. The agent receives the Java specification f_J and the buggy Python translation f_P as input. At each turn, it selects a tool from the tool suite, receives the tool's output, and incorporates that output into its accumulated context. All prior tool results are carried forward so that the agent's decisions are informed by the full history of evidence gathered so far. Once the agent judges that it has collected sufficient evidence to explain the defect, it stops invoking tools and produces a natural-language root-cause analysis. This analysis is then passed to the Patch Agent as the basis for repair.

The entire process is autonomous. The agent decides which tools to call, in what order, and when to stop. We impose an upper bound of K=10 tool invocations per defect to prevent runaway cost. In practice, no defect reaches this limit (§5.2).

3.3 Diagnostic Tool Suite

The tool suite is the mechanism by which the LLM acquires grounded, cross-lingual evidence. Table 1 summarizes all seven tools, which fall into two complementary families.

Semantic and structural differential analyzers accept f_J and f_P and expose behavioral discrepancies. `control_flow` contrasts branching, looping, and return structure. `data_flow` contrasts variable definitions, assignments, and operator usage. `call_graph` contrasts the set of callees and method invocations. Each tool extracts a lightweight semantic or structural summary from both sides and then prompts the LLM to compare the summaries and report divergences with a natural-language explanation and severity level. Framing the comparison as a differential query allows the agent to reason about the translation gap directly.

Source-context retrievers operate exclusively on the Java side and progressively widen the context available to the agent. `summarize_function` returns an LLM-generated summary of the function's purpose. `read_function_doc` extracts the JavaDoc comment of the code. `read_function_code` returns the full source of a named Java function. `read_file_code` returns the entire enclosing Java file and exposes class-level context such as fields, helper methods, and imports. This family allows the agent to escalate from a cheap summary to full file-level context only when the defect demands it.

3.4 Patch Agent

The Patch Agent synthesizes the repaired function $f_P^\star$. It receives the Java specification f_J, the buggy translation f_P, and the diagnosis. The agent operates under a system directive that establishes two invariants. First, the Java code is the authoritative specification of intended behavior. Second, the output consists of the corrected Python function only. Conditioning on diagnosis focuses the model on the specific defect identified by the Diagnosis Agent, while treating f_J as the specification allows the Patch Agent to perform a full re-translation when the existing Python is too corrupted to salvage incrementally.

3.5 Validation Pipeline

For each defect, AnchorRepair runs the Diagnosis Agent to obtain a diagnosis, invokes the Patch Agent to produce a candidate patch, and validates the result via GraalVM [31]. Specifically, the patched Python function $f_P^\star$ is spliced into the translated project, and the original Java test suite is executed against the Python code through GraalVM's cross-language interoperability. A defect is counted as repaired if and only if all relevant tests pass.

AnchorRepair performs one diagnosis-patch-validation pass per defect. A failing patch is not fed back into a new diagnosis round. This single-pass design isolates the contribution of tool-grounded diagnosis from the confounding effect of iterative refinement and bounds the cost per defect to a single generation from each agent.

4 Experimental Setup

We evaluate AnchorRepair on real translation defects and organize the study around three research questions:

- **RQ1 (Effectiveness)** How effective is AnchorRepair at repairing translation-induced defects?
- **RQ2 (Diagnosis stage)** How does the diagnosis stage drive repair success, and what tool usage patterns emerge?
- **RQ3 (Generalization)** How does AnchorRepair perform under different LLM backends, and how does it compare with a general-purpose coding agent?

4.1 Benchmark

Our benchmark is drawn from AlphaTrans [11], which provides LLM-generated Java-to-Python translations of four mature Apache Commons libraries (`commons-fileupload`, `commons-validator`, `commons-cli` and `commons-csv`). AlphaTrans provides per-method GraalVM validation results and human-written ground-truth patches. We select defects through the following procedure. First, we collect all functions whose LLM-generated translations fail GraalVM validation. This failure indicates semantic mismatch with the original Java implementation. Second, AlphaTrans itself acknowledges that GraalVM's cross-language interoperability has limitations that may cause spurious test failures unrelated to translation correctness. To eliminate such confounding cases, we re-validate the human-written ground-truth patches through the same GraalVM pipeline and retain only those defects whose ground-truth patch passes all tests. This filtering ensures that for every defect in our benchmark, a correct Python translation can be validated by the GraalVM. Any repair failure is thus attributable to the repair method rather than to infrastructure limitations.

Table 2: LLM backends evaluated in this study.

Model	Provider	Size	License
DeepSeek-V3.2 [8]	DeepSeek	671B	Open-source
Qwen3.5-Plus [7]	Alibaba	397B	Open-source
Claude-Sonnet-4.6 [5]	Anthropic	–	Closed-source

After filtering, 82 function-level defects remain. Each defect is a pair (f_J, f_P) where f_J is the correct Java function and f_P is the buggy Python translation.

4.2 Baselines

As mentioned in introduction(§ 1), code translation repair differs from traditional APR. To ensure a fair and meaningful comparison, we establish three prerequisite criteria for baseline selection: (i) the approach must have reported strong empirical results on modern program repair benchmarks; (ii) it must support Python as the target language, consistent with our translation-defect setting; and (iii) it must not rely on pre-existing test suites. Following these criteria, we surveyed recent prominent approaches and experimentally attempted those that met the basic requirements. We only successfully reproduced TransAgent [43] and D4C [37], so we choose them as our baselines.

LLM-Only. The LLM receives the correct Java function f_J and the buggy Python translation f_P in a single prompt and generates a corrected Python function directly without diagnostic tools or intermediate reasoning. This baseline isolates the contribution of tool-grounded diagnosis by holding the generation model fixed.

TransAgent [43]. TransAgent is a multi-agent translation repair pipeline that locates erroneous blocks through execution alignment and rewrites them using LLM. For our comparison, we skip its initial translation and localization stages for a fair comparison regarding its repair capability since our dataset already provides the localized translated code.

D4C [37]. D4C is a divide-and-conquer repair approach that decomposes the target function and applies iterative rewriting at the sub-function level. While D4C is designed to leverage test feedback when available, its core repair mechanism can operate effectively without test suites, making it suitable for settings where test coverage is absent or incomplete.

All baselines use DeepSeek-V3.2 [8] as the underlying LLM and are evaluated under the identical benchmark, oracle, and success criterion described below. Observed differences thus reflect repair strategy rather than backbone model capability.

4.3 Implementation

We instantiate both the Diagnosis Agent and the Patch Agent with DeepSeek-V3.2 (671B parameters, open-source) as the default backend. Within a single run of AnchorRepair, the same backend serves both agents so that comparisons attribute differences to repair strategy rather than model mixing. For generalization analysis (RQ3), we additionally evaluate with Qwen3.5-Plus [7] (397B, open-source) and Claude-Sonnet-4.6 [5] (closed-source). Table 2 summarizes the three backends.

Table 3: Repair effectiveness on 82 real Java-to-Python translation defects. DeepSeek-V3.2 is the underlying LLM. Success is judged by AlphaTrans' cross-language validator.

Method	#Fixed	Success Rate
LLM-only	28	34.15%
TransAgent [43]	17	20.73%
D4C [37]	21	25.61%
AnchorRepair	**40**	**48.78%**

The diagnosis budget is set to K=10 tool invocations per defect. The semantic and structural analyzers (control flow, data flow, call graph) use lightweight static extraction combined with an LLM comparison step. All Java-side retrieval targets are resolved from the defect's file context. AnchorRepair performs a single diagnosis-patch-validation pass per defect without iterative refinement. We report single-run results throughout.

5 Results and Analysis

5.1 RQ1: Overall Effectiveness

Table 3 reports the number of successfully repaired defects for every method. LLM-only means directly using LLM for fault analysis and patch generation. AnchorRepair achieves the best repair effectiveness. It repairs **40 out of 82** (48.78%) defects, an improvement of **12 absolute points** over LLM-only (28/82, 34.15%) and substantially more than the two dedicated translation repair pipelines (TransAgent 17/82, D4C 21/82). The comparison against LLM-only is particularly informative because LLM-only is the exact configuration our framework would degenerate to if the diagnosis stage were removed. The 12-point gap therefore attributes the improvement to the diagnosis stage.

It should be noted that both TransAgent (17/82) and D4C (21/82) repair fewer defects than LLM-only (28/82), indicating that wrapping the same LLM inside a specialized repair pipeline yields **negative** returns. TransAgent requires per-method runtime traces obtained by re-executing the source and target programs on shared unit tests. However, in the AlphaTrans setting, no such tests exist. This leads its alignment module to degenerate into unguided rewriting that frequently alters method signatures and private-field names. D4C relies on whole-function rewriting, a strategy effective for independently testable functions in Defects4J-style benchmarks. In a repository-level setting, however, it grants the model freedom to change signatures and rename Python name-mangled fields such as `self._CommandLine__options` and `self.__options`. This silently breaks cross-method contracts even when the local logic is correct. AnchorRepair avoids these failure modes by **replacing** the missing test signal with cross-lingual semantic evidence and by **exposing** class-wide context to the model before rewriting.

Finding 1: AnchorRepair improves over LLM-only by 12 points, while existing repair pipelines (TransAgent, D4C) underperform even a bare LLM prompt. These results indicate that translation-induced defects should be repaired by grounding the fix in the semantics of the original source code.

Table 4: Diagnosis-tool invocation profile of Diagnosis-Agent across 82 defects. #Defects: how many defects trigger the tool at least once; #Calls: the total number of invocations; Percentage: the proportion over all calls.

Diagnosis Tool	#Defects	#Calls	Percentage
`read_function_code`	74	104	23.11%
`read_file_code`	81	87	19.33%
`data_flow`	66	66	14.67%
`call_graph`	57	57	12.67%
`read_function_doc`	52	55	12.22%
`control_flow`	46	46	10.22%
`summarize_function`	35	35	7.78%
Total	**82**	**450**	100.00%

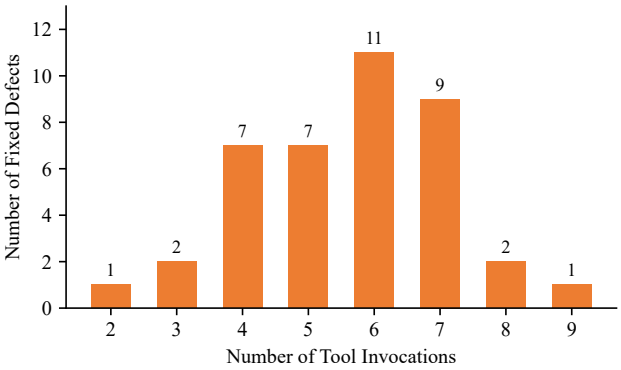


Figure 3: Number of successfully repaired defects at each diagnosis depth (tool-invocation count).

5.2 RQ2: Contribution of Each Diagnosis Stage

Every tool is actively used. Table 4 reports, for each tool, the number of defects on which it is invoked at least once (**#Defects**) and the total number of invocations (**#Calls**). Six of the seven tools are triggered on more than half of the 82 defects. The sole exception is `summarize_function` (35/82), whose lower usage is expected since modern LLMs can infer a function's semantics directly from its source code. Even this tool, however, is invoked on over 40% of defects. Since the Diagnosis Agent autonomously decides which tools to invoke (§ 3), the fact that every tool is actively exercised constitutes direct evidence of utility.

Java-side context is the dominant evidence. The two Java-side code readers dominate the invocation volume. `read_function_code` and `read_file_code` together account for 42.4% of all 450 calls, and `read_file_code` alone is triggered on 81 out of 82 defects. This pattern is consistent with the core insight of RQ1. Repository-level translation defects require class-wide context that isolated repair pipelines cannot perceive. The agent recognizes this need by fetching file-level Java evidence for nearly every defect, a behavior that specialized pipelines such as TransAgent and D4C are structurally incapable of performing.

Table 5: Repair effectiveness and cost with different LLMs and comparison with a general-purpose coding agent.

Method	Backend	#Fixed	Cost
AnchorRepair	DeepSeek-V3.2	40	$0.57
AnchorRepair	Qwen3.5-Plus	31	$0.92
AnchorRepair	Claude-Sonnet-4.6	57	$4.98
Claude Code	Claude-Sonnet-4.6	62	$10.35

Diagnosis depth. Figure 3 reports the number of successfully repaired defects at each tool-invocation depth. The distribution peaks at 6 calls (11 repairs) and concentrates in the 4–7 range, which accounts for 34 of the 40 total repairs (85%). Defects terminating after only 2–3 calls contribute just 3 repairs, indicating that shallow diagnosis rarely provides sufficient evidence for a correct fix. The Diagnosis Agent is configured with an upper bound of ten invocations. No defect reaches this limit, with the maximum observed depth being nine. The threshold therefore does not constrain the agent's investigation.

Diagnosis quality. We manually inspected the diagnosis and found most of them correctly identified both the faulty region and the error type. This suggests that the agent's tool use produces useful diagnostic evidence rather than arbitrary calls.

> **Finding 2:** All seven diagnosis tools are actively exercised, with Java-side code readers accounting for over 42% of invocations. The majority of successful repairs occur at 4–7 tool calls, and the configured upper bound of ten is never reached.

5.3 RQ3: Generalization Under Different LLMs

Table 5 reports the repair effectiveness and monetary cost of AnchorRepair under three LLM backends, together with a comparison against a general-purpose coding agent. All AnchorRepair configurations share the same diagnosis toolkit and repair pipeline. Only the underlying model differs.

All three backends surpass the LLM-only baseline (28/82). Even the smallest model, Qwen3.5-Plus, repairs 31 out of 82 (37.8%) defects, confirming that the framework contributes value regardless of model scale. It is noted that Qwen3.5-Plus incurs a higher cost than DeepSeek-V3.2. This is attributed to two factors. (i) Qwen3.5-Plus has a higher per-token output price; (2) Qwen3.5-Plus invokes tools more frequently, likely due to its weaker defect understanding that requires more exploratory calls. Replacing the backend with a stronger model yields further gains without any pipeline modification: Claude-Sonnet-4.6 repairs 57 out of 82 (69.5%) defects, a 42.5% relative improvement over DeepSeek-V3.2, at an increased cost of $4.98 versus $0.57.

Comparison with a general-purpose coding agent. We further compare against Claude Code [4], a mature agentic coding assistant powered by Claude-Sonnet-4.6 that operates without domain-specific tools. Under the same evaluation protocol, Claude Code repairs 62 out of 82 defects at a cost of $10.35. Our best configuration (AnchorRepair with Claude-Sonnet-4.6) repairs 57 out of 82 defects at $4.98. Claude Code achieves modestly higher repair

coverage (+5 defects) but at more than double the cost. This gap is explained by the absence of targeted diagnosis tools: without pre-built utilities for cross-lingual alignment, call-graph traversal, and data-flow inspection, the general-purpose agent must consume substantially more tokens reading and reasoning over raw project files. The diagnosis toolkit in AnchorRepair thus provides a favorable cost-effectiveness trade-off by concentrating the model's attention on pre-extracted evidence rather than requiring it to navigate the repository from scratch.

Finding 3: AnchorRepair generalizes across open-source and closed-source LLMs without modification. Compared to a general-purpose coding agent (Claude Code), AnchorRepair achieves comparable repair coverage at less than half the cost, demonstrating that domain-specific diagnosis tools reduce token expenditure without sacrificing effectiveness.

6 Discussion

6.1 Threats to Validity

External validity. Our benchmark comprises 82 defects from four Apache Commons libraries for the Java-to-Python language pair. While the defect count is modest, each defect is embedded in a full repository structure with cross-file dependencies, class hierarchies, and shared utility methods. This makes it substantially more complex than the isolated algorithmic functions used in snippet-level benchmarks. However, the subject projects are all utility libraries with relatively clean APIs. Results may differ for application domains with heavier concurrency, I/O, or framework-specific idioms. Additionally, all translations are produced by a single translator (AlphaTrans). Our findings may therefore not generalize to defects introduced by other translation tools.

Internal validity. LLM outputs are stochastic, and a single run may overstate or understate effectiveness. We mitigate this by holding the prompts, tool suite, oracle, and success criterion fixed across all configurations. Observed differences thus isolate the repair strategy rather than uncontrolled variation.

6.2 Limitations and Future Work

Our evaluation covers three LLM backends (two open-source, one closed-source) on a single language pair (Java-to-Python). While AnchorRepair consistently improves over the LLM-Only baseline regardless of backend, the generalizability of these findings would benefit from evaluation on additional models and larger benchmarks.

The current study focuses exclusively on Java-to-Python translation. The diagnostic tool suite leverages Java-specific artifacts such as JavaDoc and class structure. However, the underlying architecture of *differential analysis* plus *source-context retrieval* is language-agnostic in principle. Extending to other language pairs such as C++ to Rust requires adapting the retrieval tools to the source language's documentation conventions.

Despite these improvements, AnchorRepair still fails on some defects whose root cause spans multiple functions or depends on class-level invariants that cannot be localized within a single function.

We outline two future directions. First, RQ2 results reveal the Diagnosis Agent retrieves file-level Java context on 81 of 82 defects, indicating class-wide information (field declarations, inheritance structure, helper methods) is nearly always necessary for correct diagnosis. This finding suggests translation-induced defects are rarely self-contained within a single function. Proactively injecting a structured class-level summary into the initial diagnosis prompt could reduce retrieval steps while providing richer contextual grounding. Second, AnchorRepair currently repairs translated code. A natural extension is integrating diagnosis-guided repair into the translation pipeline. A closed-loop workflow could invoke differential analysis after each function translation and iterate until validation, instead of repairing afterward. However, this integration still faces a fundamental challenge that it demands high-quality per-function test oracles for real-time validation. Once this challenge is addressed, this workflow would enable early detection of systematic translation errors before they propagate to dependent functions.

7 Conclusion

We presented AnchorRepair, a dual-agent framework that repairs semantic defects in Java-to-Python code translation by treating the original Java source as an authoritative behavioral specification. A ReAct-style Diagnosis Agent autonomously invokes seven analysis tools to gather cross-lingual evidence and produce a root-cause diagnosis. A Patch Agent then uses this diagnosis to generate a corrected Python function. On 82 real translation defects from four Apache Commons projects, AnchorRepair with DeepSeek-V3.2 repairs 40 defects (48.8%), outperforming the LLM-Only baseline by 12 absolute points and two state-of-the-art translation repair methods. The framework generalizes across LLM backends without modification and achieves repair coverage comparable to a general-purpose coding agent at less than half the cost. Our results demonstrate that structured, tool-grounded diagnosis over the source-language specification is an effective strategy for automated translation repair.

Acknowledgments

This work was supported by the National Natural Science Foundation of China Grant No. 62572344 and the Beijing-Tianjin-Hebei Natural Science Foundation Cooperation Project Grant No. 25JJJJC0030.

References

[1] 2022. humaneval-x. https://huggingface.co/datasets/THUDM/humaneval-x.
[2] 2024. c2rust. https://github.com/immunant/c2rust.
[3] 2024. JavaToCSharp. https://github.com/paulirwin/JavaToCSharp.
[4] Anthropic. 2026. Claude Code: An AI-Powered Coding Assistant. https://www.anthropic.com/claude-code.
[5] Anthropic. 2026. Claude Sonnet 4.6. https://www.anthropic.com/claude/sonnet.
[6] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering 47, 9 (2019), 1943–1959.
[7] Alibaba Cloud. 2026. Qwen3.5-Plus: A Native Multimodal Large Language Model. https://www.alibabacloud.com/.
[8] DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Erhang Li, and et al. 2025. DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models. arXiv preprint arXiv:2512.02556 (Dec. 2025). arXiv:2512.02556 [cs.CL] https://arxiv.org/abs/2512.02556
[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained

model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020).
[10] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. arXiv preprint arXiv:2203.03850 (2022).
[11] 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. Proceedings of the ACM on Software Engineering 2, FSE (2025), 2454–2476.
[12] Ali Reza Ibrahimzada, Brandon Paulsen, Reyhaneh Jabbarvand, Joey Dodds, and Daniel Kroening. 2026. MatchFixAgent: Language-Agnostic Autonomous Repository-Level Code Translation Validation and Repair. arXiv:2509.16187 [cs.SE] https://arxiv.org/abs/2509.16187
[13] Jiajun Jiang, Fengjie Li, Hongyu Zhang, et al. 2025. Mapping APIs in Dynamic-typed Programs by Leveraging Transfer Learning. ACM Transactions on Software Engineering and Methodology (2025). doi:10.1145/3641848 To appear.
[14] Jiajun Jiang, Fengjie Li, Zijie Zhao, Zhirui Ye, Mengjiao Liu, Bo Wang, Hongyu Zhang, and Junjie Chen. 2025. Boosting Redundancy-Based Automated Program Repair by Fine-Grained Pattern Mining. In Proceedings of the 2025 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 85–97. doi:10.1109/ICSME64153.2025.00018
[15] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring Program Transformations From Singular Examples via Big Code. In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 255–266. doi:10.1109/ASE.2019.00033
[16] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). Association for Computing Machinery, New York, NY, USA, 298–309. doi:10.1145/3213846.3213871
[17] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 1161–1173.
[18] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the 2014 international symposium on software testing and analysis. 437–440.
[19] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. arXiv preprint arXiv:2006.03511 (2020).
[20] Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. 2025. Hybrid Automated Program Repair by Combining Large Language Models and Program Analysis. ACM Transactions on Software Engineering and Methodology (2025). doi:10.1145/3715004
[21] 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). doi:10.1145/3796507 Just Accepted.
[22] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664 (2021).
[23] Maxim Mossienko. 2003. Automated COBOL to Java recycling. In Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings. IEEE, 40–50.
[24] 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. 651–654.
[25] Vikram Nitin and Baishakhi Ray. 2024. SpecTra: Enhancing the Code Translation Ability of Language Models by Generating Multi-Modal Specifications. arXiv preprint arXiv:2405.18574 (2024).
[26] 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. 1–13.
[27] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2025. SpecRover: Code Intent Extraction via LLMs. In 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). 963–974. doi:10.1109/ICSE55347.2025.00080
[28] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In Advances in Neural Information Processing Systems, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 68539–68551. https://proceedings.neurips.cc/paper_files/paper/2023/file/d842425e4bf79ba039352da0f658a906-Paper-Conference.pdf
[29] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code translation with compiler representations. arXiv preprint arXiv:2207.03578 (2022).
[30] 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, 2 (2026), 675–690. doi:10.1109/TSE.2025.3645056
[31] Thomas Würthinger, Christian Wimmer, Christian Humer, Lukas Stadler, Doug Simon, Andreas Wöss, Matthias Grimmer, and Christian H. G. Kemper. 2011. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. In Proceedings of the 48th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '11). ACM, 1–14.
[32] 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
[33] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents. arXiv preprint (2024).
[34] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. arXiv preprint arXiv:2304.00385 (2023).
[35] Jiahong Xiang, Xiaoyang Xu, Fanchu Kong, Mingyuan Wu, Zizheng Zhang, Haotian Zhang, and Yuqun Zhang. 2024. How far can we go with practical function-level program repair? arXiv preprint arXiv:2404.12833 (2024).
[36] Yingfei Xiong, Hongyu Zhang, et al. 2021. Survey of Automatic Program Repair Techniques. Journal of Software 32, 9 (2021), 2665–2690. doi:10.13328/j.cnki.jos.006274
[37] Junjielong Xu, Ying Fu, Shin Hwei Tan, and Pinjia He. 2025. Aligning the Objective of LLM-Based Program Repair. In Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (Ottawa, Ontario, Canada) (ICSE '25). IEEE Press, 2548–2560. doi:10.1109/ICSE55347.2025.00169
[38] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Transactions on Software Engineering 43, 1 (2016), 34–55.
[39] 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://arxiv.org/abs/2405.15793
[40] 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. Proceedings of the ACM on Software Engineering 1, FSE (2024), 1585–1608.
[41] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In The Eleventh International Conference on Learning Representations (ICLR). Kigali, Rwanda. https://openreview.net/forum?id=WE_vluYUL-X
[42] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024. Thinkrepair: Self-directed automated program repair. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 1274–1286.
[43] Zhiqiang Yuan, Weitong Chen, Hanlin Wang, Xin Peng, Zhenpeng Chen, and Yiling Lou. 2026. TransAgent: Enhancing LLM-Based Code Translation via Fine-Grained Execution Alignment. Proc. ACM Softw. Eng. 3, FSE, Article FSE071 (June 2026), 23 pages. doi:10.1145/3797099
[44] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. arXiv:2404.05427 [cs.SE] https://arxiv.org/abs/2404.05427
[45] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. 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. 5673–5684.
[46] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. arXiv preprint arXiv:2206.08474 (2022).
[47] Ming-Yuan Zhu, Karthik Suresh, and Chandan K. Reddy. 2022. Multilingual Code Snippets Training for Program Translation. In AAAI Conference on Artificial Intelligence. https://api.semanticscholar.org/CorpusID:245216916

Received 2026-08-09; accepted 2026-08-26