TransAgent: Enhancing LLM-Based Code Translation via Fine-Grained Execution Alignment

ZHIQIANG YUAN, Fudan University, China
WEITONG CHEN, Fudan University, China
HANLIN WANG, Fudan University, China
XIN PENG*, Fudan University, China
ZHENPENG CHEN, Tsinghua University, China
YILING LOU, Fudan University, China

Code translation transforms code between programming languages while preserving functionality, which is critical in software development and maintenance. While traditional learning-based code translation methods have limited effectiveness due to the lack of sufficient parallel training data, Large Language Models (LLMs) have recently advanced this field with their strong code generation and comprehension capabilities. However, code translated by LLMs still suffers from diverse quality issues, such as syntax and semantic errors. In this work, we propose TransAGENT, a novel multi-agent system that eliminates the errors during LLM-based code translation. The main insight of TransAGENT is to localize error-prone code blocks via fine-grained execution alignment between source and target code. We evaluate TransAGENT on a newly constructed benchmark of recent programming tasks to mitigate data leakage. TransAGENT outperforms the latest UniTrans by up to 33.3% in translation accuracy and achieves an average improvement of 56.7% over Agentless in program repair performance. We also conduct an ablation study and evaluate TransAGENT across different LLMs, demonstrating its effectiveness and strong generalizability.

CCS Concepts: • **Software and its engineering** → **Automatic programming**.

Additional Key Words and Phrases: Code Translation, Large Language Model, Agent

ACM Reference Format:
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 (July 2026), 24 pages. https://doi.org/10.1145/3797099

1 Introduction

Code translation aims at transforming the code from one programming language (*i.e.,* source program) to another (*i.e.,* target program) while still preserving the same functionality. Code translation is prevalent in software development and maintenance, given the needs of performance optimization [1, 15], system modernization [16, 24], or technology transitions. While manually performing code translation can be time-consuming and error-prone, various automated code translation techniques (*i.e.,* transpilers) [55, 56, 60, 78] have been proposed.

*Corresponding author: Xin Peng (pengxin@fudan.edu.cn)

Authors' Contact Information: Zhiqiang Yuan, Fudan University, Shanghai, China, zhiqiangyuan23@m.fudan.edu.cn; Weitong Chen, Fudan University, Shanghai, China, 25213050003@m.fudan.edu.cn; Hanlin Wang, Fudan University, Shanghai, China, 23210240303@m.fudan.edu.cn; Xin Peng, Fudan University, Shanghai, China, pengxin@fudan.edu.cn; Zhenpeng Chen, Tsinghua University, Beijing, China, zpchen@tsinghua.edu.cn; Yiling Lou, Fudan University, Shanghai, China, yilinglou@fudan.edu.cn.

ACM 2994-970X/2026/7-ARTFSE071
https://doi.org/10.1145/3797099

Traditional rule-based code translation relies on manually written transformation rules to convert source code into target code [13, 28, 57]. However, crafting these rules requires extensive effort, and the resulting target programs often suffer from poor readability and usability [55]. To address this issue, a series of learning-based code translation methods has been proposed to enhance translation quality [55, 56, 60]. These methods train models on large amounts of parallel data (*i.e.,* paired source and target programs), enabling the models to learn translation patterns and mappings between different languages. However, high-quality parallel data for training is often scarce in practice [2, 56, 74], and the process of model training is also very time-consuming. For example, training the TransCoder model requires 32 V100 GPUs for 12 days [55].

Recent advancements in large language models (LLMs) further enhance learning-based code translation. The translation paradigm has also shifted from "train-then-translate" to "translate-then-fix", where LLMs first translate the source into the target program and then fix translation errors [49]. Translation errors in target programs fall into two categories: *syntax errors*, which violate the grammar of the target language, and *semantic errors*, where the program produces outputs that differ from the source program for the same input. Effective use of LLMs for error repair requires precise localization and concrete guidance [31, 70, 72, 83]. Syntax errors can be detected by compilers, but the suggested fixes are often not straightforward. Semantic errors are difficult to locate, as they manifest only as output discrepancies without causing compilation or runtime failures.

However, existing methods rely solely on end-to-end LLM reasoning to fix translation errors, which limits their effectiveness [49, 59, 61, 78]. For example, UniTrans, the latest LLM-based translation approach, leverages function-level test cases to guide LLMs in repairing semantic errors in the target program. Since LLMs cannot accurately access the dynamic runtime behavior of the target program [10], this prevents the detection of fine-grained logical discrepancies and the pinpointing of the exact location to repair. Consequently, UniTrans produces only a marginal accuracy gain in the error-fixing stage, increasing from 31.25% to 31.90% on the Java-to-Python task with Llama-7B [78].

To enhance LLM in code translation, particularly for fixing syntax and semantic errors in the target program, we propose TransAGENT, an LLM-based multi-agent system. *The main insight of TransAGENT is to pinpoint the error locations in the target program and provide specific fix suggestions, thus reducing the difficulty of error fixing for LLMs.* TransAGENT employs four collaborative agents: *Initial Code Translator*, *Syntax Error Fixer*, *Code Aligner*, and *Semantic Error Fixer*. First, *Initial Code Translator* generates test cases and an initial target program for the given source program; Second, *Syntax Error Fixer* captures compiler error messages, converts them into more specific fix suggestions, and leverages LLM to fix the syntax errors; Third, *Code Aligner* divides the source program into blocks using the control flow graph (CFG) and then maps each block to the target program with LLM; Lastly, based on the mapped blocks between the source and target program, *Semantic Error Fixer* localizes the error block in the target program which exhibits different runtime behaviors from its aligned block in the source program, and then uses LLMs to specifically fix the error block with the observed runtime difference.

Pinpointing semantic errors in the target program is challenging, as it requires precise knowledge of the runtime values of variables and how they deviate from their expected outputs. Intuitively, this challenge can be addressed by comparing the intermediate runtime states of the source and target programs, much like developers inspect program states to locate errors by debugging. However, identifying the correspondence between the source and target programs is challenging, as structural differences frequently arise between their implementations. Existing methods like TransMap [64] rely solely on LLMs for statement-level alignment, but this approach is fragile; for example, if a source line translates into multiple target lines or lines are shifted, LLMs often default to naive

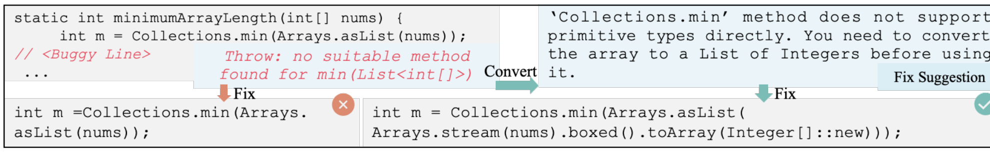

Fig. 1. Example of Fixing Syntax Errors in Target Java Program

sequential mapping, resulting in misalignments [64]. To address the mapping issue, TransAGENT introduces the *Code Aligner*, which leverages CFG-based decomposition to partition source code into atomic blocks and align them with the target program using LLMs. This block-level alignment is more robust than statement-level alignment; for instance, it groups sequential lines into a single block and maps it to the target program, effectively resolving issues such as line shifts.

Based on the aligned blocks, the *Semantic Error Fixer* simultaneously executes the source and target programs, compares the runtime states of variables within the mapped blocks, detects divergences, pinpoints the faulty block, and performs targeted repairs. Unlike traditional fault-localization [5, 29] and repair methods [5, 23] that operate on a single program, the *Semantic Error Fixer* leverages cross-program execution comparison to guide precise corrections. Moreover, unlike coarse-grained approaches such as file- or function-level localization [70, 83], TransAGENT performs fine-grained, block-level localization, where each block consists of several lines of code, to enhance LLM accuracy in correcting translation errors.

To evaluate TransAGENT, we first construct a new benchmark from recent programming tasks to mitigate the potential data leakage issue. We then evaluate the translation performance of TransAGENT, which achieves 87.1% higher accuracy than the learning-based TransCoder and 33.3% higher than the LLM-based UniTrans. An ablation study reveals that *Syntax Error Fixer* and *Semantic Error Fixer* substantially enhance translation performance, surpassing the error-fixing strategy of UniTrans. TransAGENT also demonstrates superior repair accuracy, exceeding Agentless by 56.7% and TransAGENT$_{\text{TM}}$ (with the TransMap strategy) by 14.5%. Finally, TransAGENT exhibits strong generalization to different LLMs.

We summarize our contributions as follows:

- **A Novel LLM-based Code Translation Technique.** We propose TransAGENT, an LLM-based multi-agent system for fixing the syntax and semantics errors in LLM-based code translation.
- **A Novel Code Mapping Strategy.** We design a code mapping strategy (*i.e., Code Aligner*) upon the synergy between control flow analysis and LLMs.
- **A New Code Translation Benchmark.** We construct a new code translation benchmark, which is constructed from the recent programming tasks, so as to mitigate the data leakage issue when evaluating LLM-based code translation techniques.
- **Comprehensive Evaluation.** We systematically evaluate TransAGENT across various perspectives, including the overall translation effectiveness, the ablation study of each agent, repair accuracy, and generalization.

2 Motivating Example

We illustrate the error-fixing challenges in the latest LLM-based code translation technique, UniTrans [78] via two examples. Here, we use the version of UniTrans built upon the backbone LLM Deepseek-Coder-6.7b-instruct [14]. The first example demonstrates a failure to fix a *syntax error*, while the second shows a failure to resolve a *semantic error*.

Challenges in fixing syntax errors solely with the error messages thrown by the compiler/interpreter. UniTrans relies on compiler error messages to fix syntax errors, but these messages

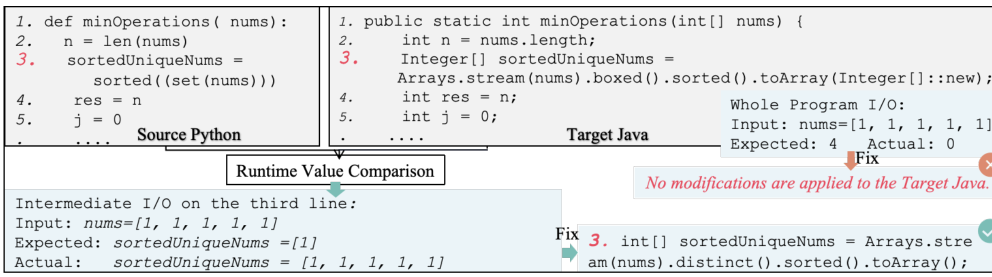

Fig. 2. Example of Fixing Semantic Errors in Target Java Program

are often vague and lack specific repair guidance, making it difficult for LLMs to resolve issues effectively. For instance, in Figure 1, the Java program `minimumArrayLength` [45], translated from Python, encounters a compilation error: `no suitable method found for min(List<int[]>)`. The error message fails to explain why the method call is invalid or how to fix it, providing limited hints for the LLMs to generate correct patches. To enhance the ability of LLMs to correct syntax errors, it is helpful to convert compiler error messages into clearer, more specific fix suggestions. For example, rephrasing the message to indicate that `Collections.min()` requires a `List` of objects rather than primitives would better guide the LLM in resolving the issue.

Challenges in fixing semantic errors solely with whole program input and output. UniTrans relies on test inputs and outputs of the entire program to address semantic errors, but this approach is often too complex for LLMs. Reasoning along full execution paths, especially with multiple logical branches, exceeds the capabilities of LLMs, as noted in recent studies [10]. For example, in Figure 2, the Java program `minOperations` [46] translated by UniTrans from Python misses a key deduplication operation in Line 3 of the source Python program. This omission causes the Java program to return an incorrect output (`0` instead of `4`). Despite using test inputs and outputs, UniTrans fails to correct this subtle logic error due to program complexity. Improving semantic error correction could involve decomposing the problem by localizing fine-grained error locations (*e.g.*, specific statements or blocks) and using relevant runtime values to fix them. For instance, comparing variable runtime values between the Python and Java programs could pinpoint the error to Line 3 in the Java program, providing more focused hints for LLMs to resolve the issue.

3 Approach

Figure 3 illustrates the workflow of TransAGENT, which is a multi-agent system consisting of four different LLM-based agents collaborating with each other, including *Initial Code Translator*, *Syntax Error Fixer*, *Code Aligner*, and *Semantic Error Fixer*.

- **Initial Code Translator** leverages test cases to enhance LLMs in translating the source program into an initial version of the target program.
- **Syntax Error Fixer** aims at iteratively addressing the syntax errors in the target program based on compilation or interpreting error messages with the self-debugging capabilities of LLMs.
- **Code Aligner** first divides the source program into blocks based on the control flow, and then leverages LLM to map each block of the source program to that of the target program. The mapping aims at facilitating a fine-grained comparison of runtime behaviors (*e.g.*, runtime value of specific variables) between the source program and the target program in the following *Semantic Error Fixer* Component. Different from previous code mapping strategies [64] which purely rely on LLMs to perform statement-level alignment, *Code Aligner is novel in incorporating the synergy of both program analysis and LLMs at the block level.*

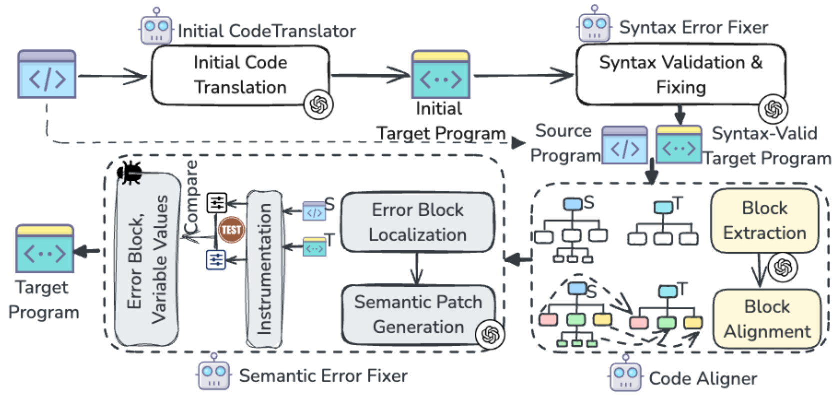

Fig. 3. Workflow of TransAGENT

- **Semantic Error Fixer** first localizes the suspicious block in the target program, which exhibits different runtime behaviors from its aligned block in the source program; and then it leverages LLMs to specifically fix the error block with the observed runtime difference. *Semantic Error Fixer is novel in fixing the semantic errors during code translation in such a fine-grained way.*

In particular, whenever the target program passes all the generated tests, the workflow terminates, and the target program is returned as the final target program; otherwise, the workflow proceeds to fix the syntax or semantic errors of the target program.

3.1 Initial Code Translator

Following the previous code translation work UniTrans [78], *Initial Code Translator* mainly includes two parts, *i.e.,* test generation and direct code translation. *Test Generation.* As revealed by UniTrans [78], including test inputs and outputs in the prompt can boost LLM-based code translation. Specifically, we first leverage LLMs to generate test inputs for the given source program with the prompt *"Please generate five inputs for the given source program"*; and the outputs of executing the source program with the generated inputs would be regarded as the test outputs. *Direct Code Translation.* We leverage LLMs to directly generate the target program (*i.e.,* the initial target program) for the given source program with the generated test inputs and outputs.

3.2 Syntax Error Fixer

Syntax Error Fixer iteratively leverages LLM to fix the syntax errors in the target program through three steps, *i.e.,* (i) *syntax validation*, (ii) *fixing strategy planning*, and (iii) *syntax patch generation*. *Syntax validation* invokes external tools (*e.g.,* compilers for Java/C++ or an interpreter for Python) to check the syntactic correctness of the target program. If no syntax errors are found, the target program proceeds to the next agents; otherwise, the process moves to *fixing strategy planning*, where error message is collected and converted into specific fix suggestions. As shown in Figure 1, the fixing suggestion is generated by the LLM based on the collected error message. The underlying inspiration is that planning can enhance the effectiveness of LLM-based agents [51]. Based on the suggestion, *syntax patch generation* prompts LLMs to create patches to fix the errors.

Fixing workflow. The patched target program would further go to *syntax validation* of the next iteration. The iterative process terminates when (i) there are no syntax errors or (ii) the same syntax errors (via String comparison) occur at the same buggy location as the previous iteration (to prevent an infinite loop). Otherwise, if there are syntax errors different from the previous iteration, TransAGENT continues the iterative fixing process.

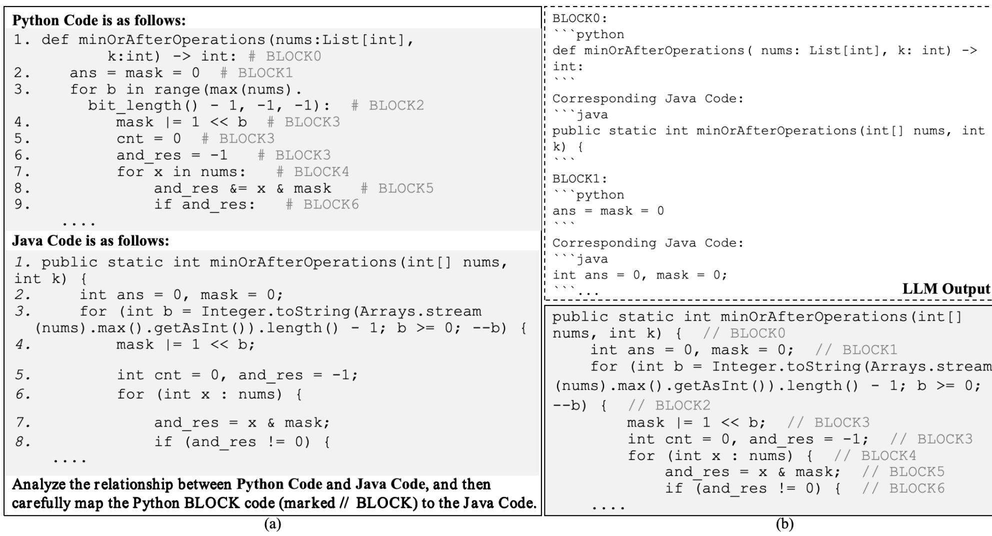

Fig. 4. Prompts in *Coder Aligner*

3.3 Code Aligner

As shown by the motivating example in Section 2, directly fixing semantic errors with test inputs and outputs can be challenging for LLMs. Mapping the semantically-equivalent code elements (*i.e.,* statements or blocks) between the source program and the target program can help localize the error code element, thus narrowing down the fixing space of semantic errors. Therefore, before running *Semantic Error Fixer*, TransAGENT first includes the LLM-based agent (*i.e., Code Aligner*) to map semantically-equivalent code elements between the source program and the target program. The previous code mapping technique TransMap [64] purely relies on LLMs to perform statement-level mapping, however statement-level mapping can be too fine-grained to be practical, as it is common for (i) one statement aligns (in the source program) with multiple statements (in the target program) or (ii) the order of statements can be very different between the source program and target program. As a result, LLMs exhibit limited mapping accuracy as revealed in the evaluation of TransMap [64]. Therefore, to address these limitations, *Code Aligner* proposes a block-level mapping technique, which aligns code elements in a coarse-grained granularity (*i.e.,* block-level) with the synergy of both program analysis and LLMs. In particular, *Code Aligner* includes two steps, *i.e.,* (i) *block extraction* that divides the source program into blocks via control-flow analysis, and (ii) *block alignment* that maps each block in the source program to the target program via LLMs. For better illustration, we denote the source program as P_S and the target program as P_T.

Block Extraction. We first construct the control flow graph of the source program and then divide the source program into blocks based on the control flow with the following division criteria.

- A continuous sequence of statements that have no jumps in or out of the middle of a block would be regarded as a block. For example, in Figure 4.a, Line 4 - 6 is a block (*i.e.,* marked as BLOCK3).
- Any control flow statement (*i.e.,* the statement that can result in different execution paths, such as `while`, `for`, `try`, or `if`) would be regarded as a block. For example, in Figure 4.a, Line 3 is a block (*i.e.,* BLOCK2) with a `for` statement; Line 9 is a block (*i.e.,* BLOCK6) with an `if` statement.

In fact, the block here is similar to the concept of *basic blocks* [58] in the control flow graph. However, the basic block is often on the granularity of a three-address instruction, which can be too fine-grained for the code translation scenario. Therefore, we adjust the scope of the basic block in this work based on the two criteria above. We alternatively refer to the blocks in the source

Algorithm 1: Error Block Localization Algorithm

Input: $P_S =< B_{S1}, B_{S2}, ...B_{Sn} >, P_T =< B_{T1}, B_{T2}, ...B_{Tn} >, \mathbb{V}_S = \{V_S^{t_k}\}, \mathbb{V}_T = \{V_T^{t_k}\}, T = \{t_1, t_2, ..., t_K\}, f_{map}, f_{id}$

Output: Error block in target program B_{Te}

- **for** t_k *in* T **do**
  - $l \leftarrow 0$;
  - **while** $l < len(V_T^{t_k})$ **do**
    - $V_S \leftarrow V_S^{t_k}[l]; V_T \leftarrow V_T^{t_k}[l]$;
    - $B_{Si} \leftarrow f_{id}(V_S); B_{Tj} \leftarrow f_{id}(V_T)$;
    - **if** $V_T == NULL$ **then**
      - **return** $f_{map}(B_{Si})$;
    - **if** $B_{Tj} \neq f_{map}(B_{Si})$ **then**
      - **if** B_{Tj} *is a control flow statement* **then**
        - **return** B_{Tj} ;
      - **else**
        - **return** $f_{map}(B_{Si})$;
    - **else**
      - **if** $Equal(V_S, V_T)$ **then**
        - **continue**;
      - **else**
        - **return** B_{Tj} ;
    - $l \leftarrow l + 1$;

program as source blocks and those in the target program as target blocks. After extraction, the source program is divided into a sequence of numbered blocks, *i.e.,* $P_S =< B_{S1}, B_{S2}, ...B_{Sn} >$ where B_{Si} denotes the source block in the source program (as shown in Figure 4.a).

Block Alignment. After dividing the source program into blocks, *Code Aligner* further leverages LLMs to map each block to the target program. As shown in Figure 4.a, LLMs are prompted to map the numbered source blocks to the corresponding target block; the top part of Figure 4.b shows the mapping outputs generated by LLMs, which are further post-processed into a structured representation (as shown in the bottom part of Figure 4.b). After the block alignment, the target program is then divided into target blocks, *i.e.,* $P_T =< B_{T1}, B_{T2}, ...B_{Tn} >$, with mapping function $f_{map}(B_{Si}) = B_{Tj}$.

3.4 Semantic Error Fixer

Based on the block-level mapping between source and target program, TransAGENT then performs a fine-grained fixing process by (i) first localizing the error target blocks by comparing the dynamic behaviors of each mapped pair of source blocks and target blocks (*i.e., Error Block Localization*) and (ii) then specifically fixing the error target block with relevant error information (*i.e., Semantic Patch Generation*). Unlike previous LLM-based code translation works [49, 78] that directly leverage LLMs to fix semantic errors without pinpointing the suspicious location, *Semantic Error Fixer* can (i) not only narrow down the fixing space by pinpointing the error target block (ii) but also provide detailed error information about the runtime values within the block rather than only providing the test inputs/outputs of the entire program. We then explain each step in detail.

3.4.1 Error Block Localization. For error block localization, TransAGENT first collects the runtime values of blocks in both source and target programs (*i.e., Runtime value collection*) and then detects the target block with different values from its mapped source block (*i.e., Runtime value comparison*).

Runtime value collection. TransAGENT first collects the runtime values of all the variables within each block for both source and target programs. In particular, TransAGENT first instruments both source and target programs by adding logging statements at the entry or exit of each block (more details are in Section 4.5); then TransAGENT executes the instrumented source and target program with each test input and collects the runtime values of all the variables within each block for both source and target programs. Specifically, the execution trace of the instrumented source program P_S with test case t_k can be denoted as a list $V_S^{t_k} =< V_{S1}^{t_k}, V_{S2}^{t_k}, ..., V_{SL}^{t_k} >$, where $V_{Sl}^{t_k}$ contains the runtime values within the l^{th} execution instance of the source block S_{Bi}, *i.e.,* $S_{Bi} = f_{id}(V_{Sl}^{t_k})$, and the function f_{id} returns the block of the execution block instance. Additionally, the runtime values of source program P_S with the entire test suite T can be denoted as $\mathbb{V}_S = \{V_S^{t_k}\}$. Similarly, the runtime values of target program P_T with the entire test suite T can be denoted as $\mathbb{V}_T = \{V_T^{t_k}\}$, where $V_T^{t_k} =< V_{T1}^{t_k}, V_{T2}^{t_k}, ..., V_{TL'}^{t_k} >$.

Runtime value comparison. As illustrated in Algorithm 1, TransAGENT then localizes the error target block by comparing the collected values of each pair of mapped blocks. The algorithm iterates the comparison over each test case t_k (Line 1). In particular, along the execution trace of the target program (Line 3), the algorithm compares each block execution instance iteratively. First, when the runtime values of the current target block do not exist (*i.e.,* indicating there is some runtime error when the target program executes the block), the algorithm returns the target block that is mapped with the current source block as the error block (Line 6 - 7). Second, if the current source block and the current target block are not mapped (Line 8), which indicates there is some mismatching introduced into the control flow, the algorithm returns the control flow statement as the error block. Third, if the current source block and the current target block are mapped and their runtime values are equal (Line 14), which indicates these two blocks are semantically equivalent, the algorithm proceeds to the next iteration; otherwise, the current target block is returned as the error block when the runtime values are not equal between the mapped pair of the source block and the target block.

3.4.2 Semantic Patch Generation. After identifying the specific error block within the target program, TransAGENT leverages LLMs to generate patches for the error target block. In particular, we include both the vanilla fixing strategy and the value-aware fixing strategy as follows.

- *Vanilla fixing strategy* prompts LLMs to fix the error target block based on static information (*i.e.,* the code of the error target block and its mapped source block);
- *Value-aware fixing strategy* prompts LLMs to fix the error target block by further providing the collected runtime values of the error target block and its mapped source block.

These two fixing strategies are complementary, as existing LLMs exhibit imperfect capabilities of reasoning the runtime behaviors of program [10]. As a result, runtime values can sometimes be helpful for LLMs to understand bug causes, especially for the cases with extreme values like data overflow (which are the cases value-aware fixing strategies can be helpful for); but sometimes runtime values can be too obscure and overwhelming to negatively limit LLMs in understanding bugs (which are the cases vanilla fixing strategies can be helpful for). Our ablation study results in Section 5.2 further confirm the complementarity between these two fixing strategies.

For example, Figure 5.a and Figure 5.b show the prompts used in the vanilla and value-aware fixing strategies, respectively. For both cases, the error block identified in the previous localization step is the code segment between the markers "−1−" and "−2−". In particular, we adopt a cloze-style fixing prompt by querying LLMs to directly re-generate the correct code (*i.e.,* "Fill in the Correct Code Here"), which is commonly used in LLM-based program repair[68, 71, 73, 82]; in addition, both fixing strategies follow Chain-of-Thought reasoning prompts with two steps, which have been shown to be effective in previous research [66, 67, 79].

Example illustration. Figure 5.a illustrates the vanilla fixing strategy, which prompts LLMs to generate the correct code for the error target block based on the mapped source block and surrounding contexts. Figure 5.b illustrates the value-aware fixing strategy. After being translated

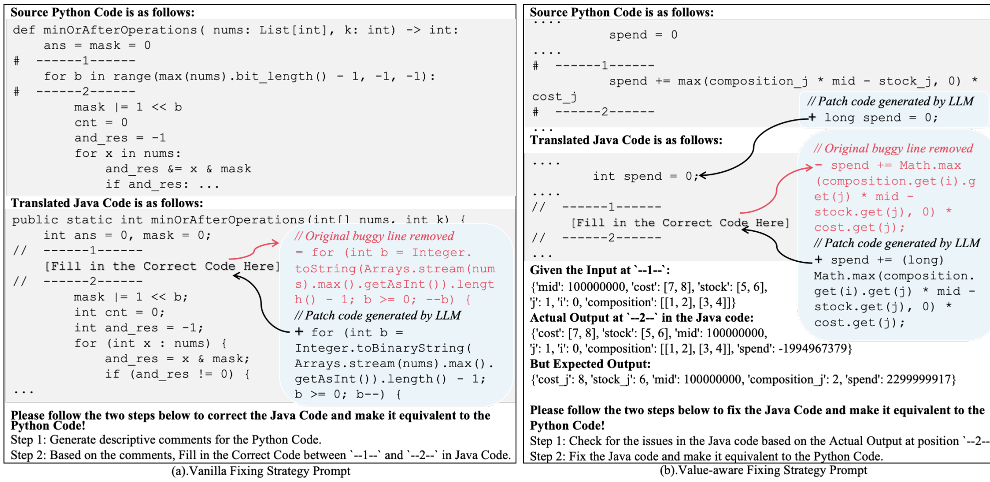

Fig. 5. Prompts in Semantic Patch Generation

from Python to Java, the target program encounters a data overflow issue where the spend variable exceeds the range for its type. The expected output is "2,299,999,917" within the mapped source block, but the actual output within the target block is "-1,994,967,379" due to the data overflow. By including such extreme runtime values in the prompt, the value-aware fixing strategy can remind LLMs of the potential type errors leading to such extreme runtime values. As a result, the value-aware fixing strategy can fix the error target block by re-declaring spend as the Long type.

Fixing workflow. During the semantic patch generation, TransAGENT iteratively applies both fixing strategies. In each iteration, each generated patch would be executed to validate whether the error block of the new target program exhibits no difference in the runtime values compared to the source program. If the runtime value difference of the current error block has been eliminated, TransAGENT proceeds to fix the next error block (if has any); otherwise, the fixing process terminates by concluding as a failed attempt.

4 Experimental Setting

We evaluate TransAGENT by answering the following research questions:

- **RQ1 (Overall Effectiveness):** How does TransAGENT compare to state-of-the-art transpilers?
- **RQ2 (Ablation Evaluation):** How does each agent in TransAGENT boost code translation?
- **RQ3 (Repair Accuracy):** How accurate is TransAGENT in repairing errors?
- **RQ4 (Generalization):** How does TransAGENT perform with different backbone LLMs?

4.1 Benchmark

Limitations of Existing Benchmarks. Although there are many existing code translation benchmarks [2, 34, 55, 56, 74], they have the following limitations. First, all existing benchmarks suffer from potential data leakage issues, as their translation tasks are constructed from public programming competitions by the training data timestamp of the most recent LLMs (*e.g.*, the most widely-used evaluation dataset TransCoder-ST [55] is created in 2020). Second,

Table 1. Line Distribution and Average Line Coverage

Line Distribution	Python (%)		Java (%)		C++ (%)	
	Pct.	Cov.	Pct.	Cov.	Pct.	Cov.
[0,5)	11.0	100.0	2.7	100.0	3.2	100.0
[5,10)	24.6	100.0	13.1	100.0	13.6	100.0
[10,15)	35.5	98.2	24.9	99.0	31.2	99.9
[15,20)	16.2	96.8	26.7	97.6	20.8	100.0
[20,60)	12.7	95.2	32.6	98.6	31.2	99.0

some benchmarks involve the translation between only two languages [3, 11, 43, 47], such as the Java-C# benchmark CodeTrans [43] or the Java-Python benchmark AVATAR [3], which limits the generalization of evaluation. Third, some benchmarks (*e.g.*, CodeNet [50]) suffer from quality issues, and half of its tasks are manually identified as incorrect code by experts [84], thus can harm the soundness of the evaluation.

New Benchmark Construction. To mitigate the aforementioned limitations, particularly the critical issue of data leakage, we establish a novel benchmark for code translation. Our selection of programming languages for this benchmark focuses on Java, Python, and C++. This choice is primarily driven by their long-standing and consistent presence among the top three most popular languages according to the TIOBE index [62]. Furthermore, this aligns with the prevailing focus of prior research in code translation [3, 34, 55, 74, 78], which predominantly concentrates on translation among these three languages. Specifically, from the programming competition websites (*e.g.*, LeetCode [35] and GeeksforGeeks [18]), we collect the solutions of programming tasks for these three programming languages, which are released after *August 2023*. As the solutions in these websites typically come with only two or three test cases, which can be insufficient for guaranteeing the semantic correctness of code [38], we further leverage gpt-4o-mini [20] to generate 10 additional test cases per solution, to ensure the sufficiency of tests for each translation task. We execute the collected code solutions with test cases and discard the tasks whose solutions exhibit inconsistent behaviors between different languages. Lastly, two authors of this work further manually check each translation task to ensure the benchmark quality.

Benchmark Statistics. In this way, we obtain 210 pairs of Python-Java translation tasks, 200 pairs of Python-C++ translation tasks, and 204 pairs of Java-C++ translation tasks. Table 1 presents the line distribution of our benchmark and its corresponding average line coverage. Furthermore, the overall average line coverage of each program with test cases achieves 98.4% for Python, 98.7% for Java, and 98.4% for C++, indicating the test sufficiency for each translation task.

4.2 Baselines

Code Translation Baselines. We include the following state-of-the-art LLM-based and learning-based transpilers as baselines.

- *UniTrans [78]* is the latest LLM-based code translator, which iteratively fixes translated programs with LLMs. It is notable that another LLM-based technique proposed by Pan et al. [49] also shares a similar fixing approach as UniTrans, thus we do not include it as a separate baseline.
- *TransCoder [55]* is a representative learning-based code translation technique, which has been evaluated by almost all the previous code translation research [2, 34, 56, 60, 74, 78].

Program Repair Baseline. To evaluate the error repair performance of TransAGENT, we compare it with *Agentless* [70], a widely evaluated LLM-based program repair approach [41, 51, 76]. We exclude AutoCodeRover [83], another commonly evaluated LLM-based method, in our comparison because its coarse-grained, function-level repair strategy is incompatible with our evaluation setting. Furthermore, to assess the impact of code mapping on repair effectiveness, we construct a variant named *TransAGENT$_{TM}$* based on TransMap. In this variant, only the *Code Aligner* component of TransAGENT is replaced with the mapping strategy from TransMap, while all other components remain unchanged.

4.3 Studied Models

In RQ4, we evaluate TransAGENT with several open-source LLMs to study its generalization. In particular, we focus on models with fewer than 10 billion parameters and whose training data cutoff predates our benchmark, to avoid data leakage.

- *Deepseek-Coder-6.7b-instruct [14]* with 6.7 billion parameters, initialized from deepseek-Coder-6.7b-base and fine-tuned on 2 billion tokens of instruction data with a knowledge cutoff of February 2023.
- *Llama-3-8B-Instruct [42]* with 8 billion parameters, which is an instruction-tuned model from the Llama-3 family, optimized for dialogue usage with a knowledge cutoff of March 2023.
- *ChatGLM2-6b [9]* with 6 billion parameters, which is the second version of ChatGLM-6B [8] released in June 2023 and fine-tuned for general-purpose tasks.

Additionally, we also include *Deepseek-Coder-33b-instruct* (33B parameters) in our experiments to evaluate the generalization of TransAGENT on larger LLMs.

4.4 Evaluation Metrics

Code Translation Metrics. Following previous work [2, 55, 56], we use the following metrics to evaluate the effectiveness of code translation techniques.

- *Computational Accuracy (CA)* [55], the most important metric in code translation, which measures translation accuracy based on functional correctness. CA assesses whether the target program passes all test cases, *i.e.,* whether the target and source programs produce the same outputs with the same test inputs.
- *CodeBLEU* [53], a metric for the similarity between target and source program.

4.5 Implementation

Baseline Implementation. For UniTrans [78], we obtain its implementation from its replication package and make the following adjustments for comparison with TransAGENT. First, we replace the backbone LLM in UniTrans with the same LLM used in TransAGENT. Second, we modify its fixing phase by splitting it into a syntax error fixer and a semantic error fixer, so as to compare with relevant components of TransAGENT. Furthermore, we align its iteration strategy (*i.e.,* only iterating within a fixed threshold of iterations) into the same dynamic strategy as TransAGENT, for fair comparison. For TransCoder [55] implementation, we directly replicate it with the released implementation with its optimal model weights. We fix the beam_size parameter at 10 and select the first output to re-evaluate it on our benchmark. For $\text{TransAGENT}_{\text{TM}}$, we replace the *Code Aligner* component with the TransMap-related implementation. For Agentless, we directly use the publicly released implementation.

TransAGENT Implementation. For each agent in TransAGENT, (i) *Initial Code Translator* adopts the same setting as UniTrans; (ii) *Syntax Error Fixer* adopts javac for Java, GCC for C++, and the Python interpreter for Python, for syntax validation; (iii) *Code Aligner* adopts Joern [30], a static code analysis tool to generate control flow graphs, which supports multiple languages; (iv) in *Semantic Error Fixer*, for *Error Block Localization*, we insert log statements at either the entry or exit points of each block to capture the runtime values of all variables within the block. If the code block contains a return statement, a log statement is inserted at the entry to capture the return value; otherwise, it is located at the exit. For *Runtime Value Comparison*, we convert the recorded variable values into "JSON" format for standardized comparison. Data types such as "List," "Array," and "Deque" are mapped to "JSON" arrays, and "int," "float," and other numeric types are converted to "JSON" numbers. Such a conversion helps standardize comparison across different programming

Table 2. Translation Effectiveness of Different Transpilers

Transpilers	Java to Python		Java to C++		C++ to Java		C++ to Python		Python to C++		Python to Java	
	CA(%)	CB	CA(%)	CB	CA(%)	CB	CA(%)	CB	CA(%)	CB	CA(%)	CB
TransCoder	12.1	29.3	13.4	43.3	41.5	47.0	24.5	31.1	10.5	36.0	2.4	40.1
UniTrans	85.0	45.3	93.0	69.1	65.5	77.3	86.0	46.5	81.8	59.7	56.2	65.8
TransAGENT	**93.2**	**46.0**	**94.0**	**69.2**	**91.0**	**80.5**	**94.5**	**47.1**	**87.4**	**59.9**	**89.5**	**69.8**

languages, which might have varying data types and structures. In addition, we also include a one-shot example in all prompts to guide LLMs in generating the required output format.

LLM Settings. For each studied open-source LLM, we use their released model and weights from HuggingFace [26]. To control randomness in our experiments, we set the parameters to "temperature=0" and "do_sample=False".

4.6 Experimental Procedure

In this section, we introduce the corresponding evaluation methodology for each research question.

RQ1 (Overall Effectiveness). RQ1 compares the overall code translation effectiveness of TransAGENT with two baselines (*i.e.,* UniTrans [78] and TransCoder [55]) in terms of CA and CodeBLEU metrics.

RQ2 (Ablation Evaluation). RQ2 evaluates the contribution of each agent in TransAGENT (except *Initial Code Translator* as it is a basic component widely used in previous code translation work [78] but not the contribution of our approach). For better illustration, we adopt the following abbreviations: *ICT* for *Initial Code Translator*, *SynEF* for *Syntax Error Fixer*, and *SemEF* for *Semantic Error Fixer*. In particular, we investigate the following variants of TransAGENT for an ablation study. (i) *ICT*: including only *Initial Code Translator* for code translation; (ii) *ICT + SynEF*: including both *ICT* and *SynEF* agents; (iii) *ICT + SynEF + SemEF*: including *ICT*, *SynEF*, and *SemEF* agents. Additionally, we further evaluate the complementarity between two fixing strategies for semantic errors (*i.e.,* vanilla and value-aware fixing strategies) with the following variants: (iv) *ICT + SynEF + Val*, which applies the value-aware fixing strategy to the *ICT + SynEF* variant, and (v) *ICT + SynEF + Val + Van*, which further applies the vanilla fixing strategy to the *ICT + SynEF + Val* variant.

RQ3 (Repair Accuracy). RQ3 assesses the program repair performance of TransAGENT in comparison with Agentless and $\text{TransAGENT}_{\text{TM}}$. Given an initial target program containing translation errors, generated by the *Initial Code Translator*, we apply each approach to repair it. A repair succeeds if the resulting target program passes all test cases; otherwise, it fails. We use CA as the evaluation metric.

RQ4 (Generalization Evaluation). RQ4 replaces the default backbone LLM (*i.e.,* Deepseek-Coder-6.7b-instruct) in TransAGENT with three different LLMs (*i.e.,* Llama-3-8B-Instruct, ChatGLM2-6b and Deepseek-Coder-33b-instruct) to evaluate the generalization of TransAGENT.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Overall Effectiveness

Table 2 shows the translation performance of TransAGENT and baselines. Overall, TransAGENT achieves the best performance across all six translation scenarios, particularly for translations between dynamic and statically typed languages. For example, in the Python-to-Java translation task, TransAGENT outperforms TransCoder in CA by 87.1% (= 89.5% - 2.4%) and UniTrans by 33.3% (= 89.5% - 56.2%). In addition, we also conduct the Wilcoxon signed-rank test [54] to assess the statistical significance of the performance differences. The results yield $p \ll 0.001$ against both UniTrans and TransCoder, indicating that TransAGENT significantly outperforms the baselines.

Our manual analysis of the translation results of the TransAGENT and baselines (TransCoder or UniTrans) shows that there are no cases in which the baselines (TransCoder or UniTrans) succeed while TransAGENT fails. This result underscores the inherent limitations of learning-based approaches (*i.e.,* TransCoder), which rely on data-driven learning and are fundamentally constrained by the quality and coverage of their training data. In contrast, LLM-based methods like TransAGENT and UniTrans possess broader generalization capabilities. However, TransAGENT and UniTrans differ in their error correction strategies. TransAGENT adopts a two-step process, first localizing faults and then performing targeted repairs, while UniTrans attempts to fix errors directly without explicit localization, relying solely on its end-to-end capabilities. This lack of precise fault localization ultimately limits the overall translation performance of UniTrans.

To better understand the underlying causes of these differences, we conduct a qualitative analysis of cases where the baselines fail while TransAGENT succeeds. For TransCoder, frequent failures arise from either duplicating source language syntax in the target language or mismatching APIs, two bug categories summarized by [49] that are not observed in TransAGENT. For UniTrans, the failed translations exhibit two patterns, including modifying code lines unrelated to the root cause of the error and returning the same program as the buggy translated program [78]. One potential reason is that, without fine-grained fault localization, UniTrans faces challenges in accurately modifying the buggy program based solely on test inputs and outputs. For example, as shown in Figure 2, UniTrans fails to resolve the error because it fails to identify that the fault originates from the third line of the program. In contrast, TransAGENT accurately detects the fault by analyzing runtime execution to pinpoint the exact error location, and then leveraging an LLM to perform a precise repair.

Cost Evaluation. We collect the number of iterations and average time costs for TransAGENT. Table 3 presents the distribution of iteration counts across different translation tasks. In over 92% of cases, TransAGENT completes repair in no more than two iterations, indicating low iteration overhead. Additionally, a time consumption analysis shows that TransAGENTtakes an average of 19 seconds per example, while UniTrans takes 24 seconds, highlighting the greater efficiency of TransAGENT.

Table 3. Iteration Proportions in TransAGENT

#Iteration	0	1	2	[3,4]
Java to Python	89.5%	7.5%	1.5%	1.5%
Java to C++	93.3%	1.6%	0.5%	4.7%
C++ to Java	89.0%	4.2%	1.6%	5.2%
C++ to Python	92.7%	5.2%	2.1%	0.0%
Python to C++	88.5%	5.7%	2.1%	3.7%
Python to Java	80.9%	6.9%	4.4%	7.8%

Translation Effectiveness Evaluation Summary: TransAGENT is an effective approach for code translation tasks, consistently outperforming existing state-of-the-art transpilers. Meanwhile, TransAGENT incurs relatively low iteration and time costs.

5.2 RQ2: Ablation Evaluation

Table 4 illustrates the contribution of each agent in UniTrans and TransAGENT to the translation performance. Overall, the *Syntax Error Fixer* and *Semantic Error Fixer* components in TransAGENT positively contribute to translation performance. For example, TransAGENT achieves a 31.0% improvement over the *Initial Code Translator* in the C++-to-Java task and a 40.0% improvement in the Python-to-Java scenario.

Both the *Syntax Error Fixer* and the *Semantic Error Fixer* in TransAGENT yield larger gains in translation performance than their counterparts in UniTrans. For example, in the Python-to-Java task, the *Syntax Error Fixer* in TransAGENT outperforms that in UniTrans by 27.2% (=32.9%-5.7%),

Table 4. Performance Comparison of UniTrans (UT) and TransAGENT (TA) Agents

CA (%)	Java to Python		Java to C++		C++ to Java		C++ to Python		Python to C++		Python to Java	
	UT	TA	UT	TA	UT	TA	UT	TA	UT	TA	UT	TA
ICT	84.5		89.1		60.0		83.0		76.8		49.5	
ICT+SynEF	**85.0**	**85.0**	**91.0**	**91.0**	64.8	**87.9**	86.0	**87.5**	79.0	**81.3**	55.2	**82.4**
ICT+SynEF+SemEF	85.0	**93.2**	93.0	**94.0**	65.8	**91.0**	86.0	**94.5**	81.0	**87.4**	56.2	**89.5**
$\Delta_{\text{ICT+SynEF/Val}}\%$	-	6.3	-	2.5	-	2.1	-	6.5	-	4.6	-	4.3
$\Delta_{\text{ICT+SynEF+Val/Van}}\%$	-	1.9	-	0.5	-	1.0	-	0.5	-	1.5	-	2.8

Table 5. Repair Performance Comparison of Agentless, TransAGENT$_{\text{TM}}$ and TransAGENT

CA (%)	Java to Python	Java to C++	C++ to Java	C++ to Python	Python to C++	Python to Java
Agentless	9.3	18.2	2.5	20.6	2.2	0.9
TransAGENT$_{\text{TM}}$	37.5	31.8	77.2	55.9	40.9	63.2
TransAGENT	**59.4**	**45.4**	**83.5**	**76.5**	**50.0**	**79.2**

while the *Semantic Error Fixer* shows a 6.1% (=7.1%-1.0%) improvement. The substantial performance gains of the *Syntax Error Fixer* in TransAGENT over UniTrans stem from its planning strategy for handling error messages, while the advantage of the *Semantic Error Fixer* lies in its fine-grained fault localization prior to repair.

The *Semantic Error Fixer* in TransAGENT is more effective at repairing translation errors between dynamic and static languages than between dynamic languages. For example, in the Java-to-C++ scenario, the *Semantic Error Fixer* improves the performance of TransAGENT by 3.0%, whereas in the Python-to-Java scenario it delivers a 7.1% improvement. This difference arises from the structural gaps between dynamic and static languages. Dynamic languages such as Python often employ compact constructs, for example, “stnum = "".join(sorted(str(num)))”, which combine multiple operations (*e.g., type conversion*, *string manipulation*, *sorting*, and *aggregation*) into a single line. When performing program translation, LLMs struggle to disentangle the densely packed operations in such expressions, often resulting in semantic mismatches. In this case, the `sorted` function in Python defaults to ascending order, but the *ICT* incorrectly translates it into `Arrays.sort(.., Collections.reverseOrder())`, which produces descending order and breaks semantic consistency. To address such errors, the *Semantic Error Fixer* compares runtime states to identify inconsistencies. During repair, it then provides test cases (*e.g.,* input `num = 2134` with expected output `1234`) that reveal the correct behavior, enabling the LLM to generate accurate fixes.

Both Value-aware and Vanilla fixing strategies in *Semantic Error Fixer* can improve the overall performance of TransAGENT in code translation. For instance, in the Java-to-Python translation scenario, the Value-aware strategy increases CA by 6.3% over *ICT + SynEF*, while the Vanilla strategy provides an additional 1.9% improvement. This shows that the two fixing strategies are complementary, enhancing the ability of TransAGENT to correct errors and improve translation performance.

Ablation Evaluation Summary: Both the *Syntax Error Fixer* and the *Semantic Error Fixer* in TransAGENT contribute positively to improving the translation performance of LLMs. Moreover, these components are more effective than their counterparts in UniTrans.

5.3 RQ3: Repair Accuracy

Table 5 presents the program repair performance of TransAGENT and the baselines. The results show that TransAGENT consistently outperforms both Agentless and TransAGENT$_{\text{TM}}$. Overall, TransAGENT achieves an average repair accuracy that is 56.7% higher than Agentless

```
1.int maximumSetSize(vector<int> &nums1,
   vector<int> &nums2) {
2.  unordered_set<int> set1(nums1.begin(),
      nums1.end());
3.  unordered_set<int> set2(nums2.begin(),
      nums2.end());
4.  int common = 0;
5.  for (int x: set1){
6.     common += set2.count(x);} ....
```

Source C++ Program

```
1.public static int maximumSetSize(int[]
    nums1, int[] nums2) {
2.  HashSet<Integer> set1 = new HashSet<>();
3.  HashSet<Integer> set2 = new HashSet<>();
4.  for (int num : nums1) {
5.     set1.add(num);}
6.  for (int num : nums2) {
7.     set2.add(num); }
8.  int common = 0;
9.  for (int x : set1) {
10.     if (set2.contains(x)) {
11.       common++;}} ....
```

Target Java Program

TransMap: `1→1; 2→2; 3→3; 4→4; 5→5; 6→6;` ....

Code Aligner: `[1]→[1]; [2,3,4]→[2,3,4,5,6,7,8]; [5]→[9];[6]→[10,11];` ...

Mapping Result

Fig. 6. Mapping Example: TransMap vs. *Code Aligner*

and 14.5% higher than TransAGENT$_{\text{TM}}$, which employs the TransMap mapping strategy. The Wilcoxon Signed-Rank Test [54] yields $p = 1.5 \times 10^{-58}$ versus Agentless and $p = 1.0 \times 10^{-4}$ versus TransAGENT$_{\text{TM}}$, indicating that TransAGENT significantly outperforms the baselines in error repair ($p \ll 0.001$). This advantage stems from two key innovations: a hybrid code mapping strategy that integrates program analysis with LLMs for source-to-target alignment, and a fault localization method that leverages intermediate runtime states to identify and fix errors.

For code mapping, the *Code Aligner* in TransAGENT employs a hybrid strategy that combines CFG analysis with LLM-based alignment. It first decomposes the source program into semantically coherent code blocks and then aligns them using the LLM, effectively handling complex scenarios such as line shifts. For example, as shown in Figure 6, the *Code Aligner* groups C++ lines 2–4 as a logical block and accurately maps them to Java lines 2–8, demonstrating robustness to non-sequential and interleaved mappings. In contrast, TransMap in TransAGENT$_{\text{TM}}$ relies solely on sequential LLM-based alignment, which is prone to mapping errors in Figure 6, for instance, by incorrectly aligning line 5 in the source with line 5 in the target, thereby reducing the effectiveness of fault localization and repair.

For fault localization, TransAGENT analyzes the intermediate runtime states of both the source and target programs, enabling it to precisely identify the root causes of errors rather than relying on coarse-grained error signals. In contrast, Agentless depends solely on end-to-end LLM reasoning for both localization and repair. When only high-level error information is available, Agentless must infer the root cause directly, which is challenging for current LLMs and results in substantially lower repair accuracy.

> **Repair Accuracy Evaluation Summary:** TransAGENT achieves substantial program repair performance by combining hybrid code mapping with runtime-guided fault localization, effectively overcoming key limitations of existing LLM-based repair methods.

5.4 RQ4: Generalization Evaluation

Table 6 presents the translation performance of each component in TransAGENT across different LLMs. The results indicate that TransAGENT generalizes effectively, consistently enhancing the

Table 6. Generalization of TransAGENT in Llama-3-8B-Instruct (L3), ChatGLM2-6b (CG), and Deepseek-Coder-33b-instruct (D33)

CA (%)	Java to Python			Java to C++			C++ to Java			C++ to Python			Python to C++			Python to Java		
	L3	CG	D33	L3	CG	D33	L3	CG	D33	L3	CG	D33	L3	CG	D33	L3	CG	D33
ICT	72.9	18.4	93.2	75.1	7.5	89.3	53.5	6.0	78.0	79.5	16.5	94.0	66.5	4.5	89.0	29.7	5.4	72.2
ICT+SynEF	78.7	19.3	93.7	81.1	14.4	95.4	70.5	7.0	95.0	83.5	17.0	95.5	75.5	6.0	92.0	55.0	7.2	86.1
ICT+SynEF+SemEF	87.9	48.8	96.6	86.6	20.4	95.9	77.0	11.5	96.0	90.5	55.0	98.5	78.5	13.5	93.5	61.2	14.9	91.9

code translation performance of all LLMs. Even for the large-scale Deepseek-Coder-33b-instruct, TransAGENT provides substantial gains. For example, in the Python-to-Java task, TransAGENT improves performance by 31.5% (=61.2%-29.7%) with Llama-3-8B-Instruct, by 9.5% (=14.9%-5.4%) with ChatGLM2-6b, and by 19.7% (=91.9%-72.2%) with Deepseek-Coder-33b-instruct.

Both the *Syntax Error Fixer* and the *Semantic Error Fixer* demonstrate strong adaptability across different LLMs, contributing to consistent improvements in translation performance. The *Syntax Error Fixer* achieves gains of 25.3% with Llama-3-8B-Instruct in Python-to-Java and 6.9% with ChatGLM2-6b in Java-to-C++. Similarly, the *Semantic Error Fixer* yields improvements of 9.2% with Llama-3-8B-Instruct in Java-to-Python and 38.0% with ChatGLM2-6b in C++-to-Python. These results indicate that each component of TransAGENT generalizes effectively across models, thereby enhancing overall translation performance.

> **Generalization Evaluation Summary:** TransAGENT can be applied to different LLMs to enhance their performance in code translation tasks.

6 Discussion

6.1 Translation Performance Impact Analysis

Code Complexity Impact. We analyze how source code complexity, measured by lines of code (LOC) and cyclomatic complexity (CC), affects the translation performance of TransAGENT and baselines, as shown in Table 7. *First, the translation performance of TransAGENT remains stable as LOC or CC increases.* For instance, in the LOC group from [31, 45] to [46, 60], TransAGENT's CA increases from 84.6% to 100.0%, and in the CC group from [11, 15] to [16, 20], CA increases from 89.5% to 100.0%. *Second, for the same ranges of LOC or CC, TransAGENT consistently outperforms the baseline.* For example, in the LOC range [31, 45], TransAGENT achieves a CA that is 78.4% higher than TransCoder and 7.7% higher than UniTrans. This performance advantage primarily results from TransAGENT performing fine-grained fault localization by comparing intermediate runtime states between the source and target programs, which enables precise correction of translation errors. In contrast, UniTrans and TransCoder rely solely on end-to-end semantic reasoning and cannot exploit the intermediate states of the target program. Consequently, they fail to address subtle discrepancies that emerge only at runtime, such as precision-related issues [78]. As illustrated in Figure 5.a, the translated program incorrectly infers the type of the variable `spend` as `int` instead of the expected `long`. Neither UniTrans nor TransCoder can correct this error. In contrast, by analyzing and comparing the intermediate states of the source and target programs, TransAGENT detects the divergence in the runtime values of `spend`, pinpoints the faulty statement, and leverages the mismatch between the observed and expected outputs to perform precise correction.

TransAGENT demonstrates stable performance on complex function-level code and thus shows promising potential for project-level extension. Existing project-level translation approaches generally adopt a decompositional paradigm, in which a project is first decomposed into smaller units (typically functions) and then translated in a bottom-up order following the call graph [7, 21, 27]. Under this paradigm, TransAGENT can be naturally integrated into project-level workflows by being

Table 7. Translation Performance (CA%) of TransAGENT and Baselines across Line of Code (LOC) and Cyclomatic Complexity (CC)

LOC Groups	TransCoder	UniTrans	TransAGENT	CC Groups	TransCoder	UniTrans	TransAGENT
(1,15]	19.8	75.6	91.9	(1,5]	20.3	75.8	92.1
[16,30]	16.0	81.1	92.2	[6,10]	13.4	81.3	90.8
[31,45]	6.2	76.9	84.6	[11,15]	4.2	79.2	89.5
[46, 60)	0.0	84.6	100.0	[16, 20)	0.0	83.3	100.0

Table 8. Translation Performance on Existing Benchmark

CA(%)	Java to Python	Java to C++	C++ to Java	C++ to Python	Python to C++	Python to Java
TransCoder	51.0	85.5	71.2	46.0	63.6	37.8
SDA-Trans	58.1	84.7	66.3	55.4	41.2	42.6
UniTrans	92.5	65.8	94.0	92.3	59.0	89.8
TransAGENT	95.7	98.0	95.2	94.7	96.2	93.3

applied to individual functions during translation, thereby providing fine-grained runtime feedback and repair at the unit level. By improving per-function correctness and stability, TransAGENT enhances the overall reliability of project-level translation.

Cross-Language API Mapping Impact. TransAGENT performs cross-language core API mapping by leveraging LLMs' pre-trained knowledge of standard library APIs across different programming languages. Since LLMs are pretrained on large-scale multilingual code and documentation corpora, they exhibit strong capabilities in understanding common APIs [12, 65, 69] (*e.g.,* Python's `math` and `collections`, Java's `java.io.*` and `java.lang.*`, and C++'s `<iostream>` and `<queue>`) within given code contexts. For instance, in our benchmark, TransAGENT successfully identifies the `count()` function in the Python expression `(nums.count(m)+1)//2` and correctly maps it to Java's `java.util.Collections.frequency()` as `(Collections.frequency(nums,m)+1)/2`. In addition, during the error-fixing stage, TransAGENT guides LLMs to correct mistranslations in API usage. For example, in the statement "stnum = "".join(sorted(str(num)))", the densely packed operations cause LLMs to incorrectly translate `sorted` function as `Arrays.sort(.., Collections.reverseOrder())`. In the error-fixing stage, supplying `num=2134` with the expected output `stnum=1234` guides LLMs to produce the correct translation, `Arrays.sort(numChars)`.

Table 9. Comparison of Block Mapping Accuracy Between TransAGENT and TransMap

Mapping Accuracy(%)	Java to Python	Java to C++	C++ to Java	C++ to Python	Python to C++	Python to Java
TransMap	64.6	79.2	58.3	80.2	93.8	67.7
TransAGENT	95.8	97.9	97.9	99.0	100.0	96.9

Block Mapping Impact. Block-level mapping plays a critical role in improving translation correctness by providing fine-grained structural alignment between source and target programs. To assess the reliability of the mapping results, we compare TransAGENT with TransMap via manual inspection. In particular, we manually inspect 290 mapped source–target block pairs (95% confidence level, 0.05 margin of error) and report the mapping accuracy in Table 9. Overall, TransAGENT achieves 95.8%–100% mapping accuracy across all six language pairs, substantially outperforming TransMap (64.6%–93.8%). This accuracy gap indicates that TransAGENT provides more reliable alignment signals, which are crucial for guiding the subsequent error fixing. Moreover, occasional mapping failures are unlikely to mislead the subsequent repair process. This is because TransAGENT compares only the runtime values of variables with the same names within aligned blocks, thereby naturally limiting the impact of imperfect alignments. For example, between Python's "`a,b=0,1`" and Java's "`a=0`", only the shared variable "`a`" is used for runtime comparison, preventing unintended mismatches.

In addition, we compare TransAGENT with BatFix [52], which matches source and target control-flow graphs via a MaxSAT-based formulation. Since BatFix only supports Java-to-C++ and Python-to-C++, we conduct comparisons on these two language pairs. Overall, TransAGENT achieves substantially higher mapping accuracy than BatFix, reaching 97.0% vs. 73.0% on Java-to-C++ and 98.0% vs. 32.0% on Python-to-C++.

6.2 Existing Benchmark Evaluation

In the experiments, we collect a new dataset to assess the translation performance of TransAGENT and the baselines, mitigating potential data leakage issues. To further evaluate the generalization of TransAGENT across different datasets, we use the benchmark released by Roziere et al.[55], which is widely evaluated in prior work [2, 11, 32, 55, 56, 78]. Additionally, we include a new baseline, SDA-Trans [36], which is a learning-based method augmented with symbolic analysis. Table 8 presents the translation performance of TransCoder, SDA-Trans (using the results reported in the original paper, as the reproduction packages are unavailable [36]), UniTrans, and TransAGENT on this benchmark. As shown in the table, TransAGENT consistently outperforms all baselines on this benchmark, highlighting its robustness and generalization ability. Furthermore, TransAGENT achieves a CA of up to 98.0% on this dataset, translating nearly all programs correctly. This highlights the necessity of constructing a new evaluation dataset to reliably assess translation performance.

7 Threats to Validity

We categorize threats to the effectiveness of this work into internal effectiveness and external effectiveness. (i) One threat to validity is potential bugs in the code implementation of TransAGENT, which could lead to translation failures. To mitigate this, we use instances of translation failures to debug and improve the implementation. (ii) Another threat is potential data leakage due to overlap between the evaluation dataset and the training data of LLMs. To address this, we manually construct a new dataset from a time frame after the knowledge cutoff date of LLM, specifically after *August 2023*. We also design comprehensive test cases and calculate line coverage to ensure the equivalence of the constructed source and target code. (iii) Another threat to internal effectiveness is that TransAGENT assumes the source program is executable, as it relies on runtime feedback to perform block-level alignment and to validate repairs. If the source code is not runnable (*e.g.,* missing dependencies, incomplete environments, or uncompilable projects), TransAGENT may fail to obtain reliable execution signals and thus cannot proceed. This executability assumption is also common in execution-guided translation/repair approaches [49, 59, 64, 78]. To mitigate this threat in our evaluation, we only include instances whose source programs can be executed in a controlled environment. Regarding test data, TransAGENT does not rely on pre-existing test suites, as it can automatically generate additional tests when needed. (iv) Another threat involves the reproduction of baseline and the calculation of metrics. To minimize these threats, we strictly follow the reproduction documentation and use the source code provided for the baseline. All baselines are evaluated consistently on our collected dataset, and we reuse the code provided in [2] for implementing evaluation metrics like CodeBLEU.

8 Related Work

8.1 Code Translation

Early code translation research relied on program analysis techniques with manually formulated rules, such as C2Rust [6] for translating C into Rust, and Sharpen [57] and JavaSharp [28] for converting Java to C#. However, these methods are time-intensive and limit the accuracy and readability of the translated programs [55]. Learning-based approaches have since emerged [32, 48], significantly improving over rule-based methods but still struggling with data scarcity and the time-consuming training process [60, 78, 84].

LLMs have shown promise in software engineering tasks [37], such as code generation [25, 39, 77], program repair [17, 22, 51, 72], and code summarization [4, 19]. Pan et al. [49] identify key syntactic and semantic challenges in LLM-based code translation. Therefore, adopting the "translate-then-fix" paradigm is an effective way to enhance LLM-based translation performance. UniTrans [78], a state-of-the-art method, adopts this paradigm to conduct code translation. However, the error-fixing stage suffers from poor performance due to the lack of detailed runtime error information. TransMap improves the manual fixing process by aligning source and target code, but this alignment is purely statement-level and solely relies on the LLM, without structural or semantic guidance [64]. As a result, this mapping approach is fragile in complex mapping scenarios; for example, a single source line may translate into multiple target lines, or lines may be shifted, ultimately compromising the effectiveness of error repair. To address these issues, we propose TransAGENT. In the mapping phase, TransAGENT employs CFG-based decomposition to partition the source program into atomic blocks, each forming an independent structural unit. LLMs then map these blocks to the target program. This hybrid approach improves mapping robustness and mitigates the limitations of TransMap. In the fixing phase, TransAGENT compares the intermediate execution states of the source and target programs to detect semantic discrepancies. This runtime-based localization offers clearer repair cues, substantially improving error correction accuracy and overcoming the limitations of UniTrans.

8.2 Program Repair

Program repair involves fixing bugs in the code. Early methods relied on heuristic approaches [80], constraint-based techniques [75], and pattern-based methods [33, 40]. However, these approaches, which depend on manually designed templates, struggle with generalizing to diverse repair tasks. Learning-based methods like CoCoNut [44] and Tare [85] improve performance by learning bug-fixing patterns from large datasets [81]. Recent LLM-based methods such as RING [31], AutoCodeRover [83], and Agentless [70] perform program repair using static information, such as issue descriptions and code similarity. However, these approaches are primarily designed for single-program scenarios within a single language, making them ill-suited for code translation tasks that involve reasoning across both source and target programs in different languages. In contrast, TransAGENT leverages dynamic information by analyzing the runtime behaviors of basic blocks partitioned via control-flow analysis in both the source and target programs. This allows for more precise fault localization and significantly improves repair performance. Our empirical results further demonstrate that TransAGENT outperforms Agentless in the context of code translation error correction.

9 Conclusion

In this paper, we introduce TransAGENT, a novel multi-agent system to improve LLM-based code translation with fine-grained execution alignment. The experimental results show that TransAGENT outperforms the latest LLM-based code translation technique UniTrans in both translation effectiveness and efficiency; additionally, our ablation study demonstrates the contribution of each agent, and the error repair evaluation shows that TransAGENT exhibits superior error-repair capabilities compared with Agentless; lastly, we further evaluate the generalization of TransAGENT across different LLMs.

10 Data Availability

Our data and code are included in our replication package [63].

Acknowledgments

This work is supported by the National Key R&D Program of China (Grant No. 2023YFB4503805) and the National Natural Science Foundation of China (Grant No. 62332005).

References

[1] 2021. Transforming Monolithic Applications to Microservices with Mono2Micro. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 3–3. doi:10.1109/ASE51524.2021.9678878

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2023. Summarize and Generate to Back-translate: Unsupervised Translation of Programming Languages. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2023, Dubrovnik, Croatia, May 2-6, 2023*. Association for Computational Linguistics, 1520–1534.

[3] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2023. AVATAR: A Parallel Corpus for Java-Python Program Translation. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*. Association for Computational Linguistics, 2268–2281.

[4] Toufique Ahmed and Premkumar T. Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 177:1–177:5. doi:10.1145/3551349.3559555

[5] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. USENIX Association, 235–252. https://www.usenix.org/conference/usenixsecurity20/presentation/blazytko

[6] C2Rust. 2024. (2024). /urlhttps://github.com/immunant/c2rust.

[7] 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. *CoRR* abs/2503.17741 (2025). arXiv:2503.17741 doi:10.48550/ARXIV.2503.17741

[8] ChatGLM. [n. d.]. ([n. d.]). https://github.com/THUDM/ChatGLM-6B.

[9] ChatGLM2. [n. d.]. ([n. d.]). https://github.com/THUDM/ChatGLM2-6B.

[10] Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2024. Reasoning Runtime Behavior of a Program with LLM: How Far Are We? *arXiv e-prints* (2024).

[11] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree Neural Networks for Program Translation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net.

[12] Yujia Chen, Cuiyun Gao, Muyijie Zhu, Qing Liao, Yong Wang, and Guoai Xu. 2024. APIGen: Generative API Method Recommendation. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland, March 12-15, 2024*. IEEE, 171–182. doi:10.1109/SANER60148.2024.00025

[13] cxgo: C to Go transpiler. 2024. (2024). https://github.com/gotranspile/cxgo.

[14] Deepseek. [n. d.]. ([n. d.]). https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct.

[15] Roberto Rodriguez Echeverria, Fernando Macias, Victor Manuel Pavon, Jose Maria Conejero, and Fernando Sanchez Figueroa. 2015. Legacy Web Application Modernization by Generating a REST Service Layer. *IEEE Latin America Transactions* 13, 7 (2015), 2379–2383. doi:10.1109/TLA.2015.7273801

[16] Mahdi Fahmideh, Farhad Daneshgar, Ghassan Beydoun, and Fethi A. Rabhi. 2020. Challenges in migrating legacy software systems to the cloud an empirical study. *CoRR* abs/2004.10724 (2020). arXiv:2004.10724

[17] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. 1469–1481. doi:10.1109/ICSE48619.2023.00128

[18] geeksforgeeks. [n. d.]. ([n. d.]). https://www.geeksforgeeks.org/.

[19] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2023. An Empirical Study on Using Large Language Models for Multi-Intent Comment Generation. *CoRR* abs/2304.11384 (2023). arXiv:2304.11384 doi:10.48550/ARXIV.2304.11384

[20] gpt-4o mini. 2024. (2024). https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/.

[21] Ziqi Guan, Xin Yin, Zhiyuan Peng, and Chao Ni. 2025. RepoTransAgent: Multi-Agent LLM Framework for Repository-Aware Code Translation. arXiv:2508.17720 [cs.SE] https://arxiv.org/abs/2508.17720

[22] Yaoqi Guo, Ying Xiao, Jie M Zhang, Mark Harman, Yiling Lou, Yang Liu, and Zhenpeng Chen. 2026. EET: Experience-Driven Early Termination for Cost-Efficient Software Engineering Agents. *arXiv preprint arXiv:2601.05777* (2026).

[23] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. AAAI Press, 1345–1351. doi:10.1609/AAAI.V31I1.10742

[24] Sindre Grønstøl Haugeland, Phu Hong Nguyen, Hui Song, and Franck Chauvel. 2021. Migrating Monoliths to Microservices-based Customizable Multi-tenant Cloud-native Apps. In *47th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2021, Palermo, Italy, September 1-3, 2021*. IEEE, 170–177.

[25] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xin Peng, Xiwei Xu, and Qinghua Lu. 2024. FQN Inference in Partial Code by Prompt-tuned Language Model of Code. *ACM Trans. Softw. Eng. Methodol.* 33, 2 (2024), 31:1–31:32. doi:10.1145/3617174

[26] huggingface. [n. d.]. ([n. d.]). https://huggingface.co.

[27] 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 (2025), 2454–2476. doi:10.1145/3729379

[28] JavaToCSharp. 2024. https://github.com/paulirwin/JavaToCSharp.

[29] Wei Jin and Alessandro Orso. 2013. F3: fault localization for field failures. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*. ACM, 213–223. doi:10.1145/2483760.2483763

[30] joern. [n. d.]. ([n. d.]). https://docs.joern.io/server/.

[31] Harshit Joshi, José Pablo Cambronero Sánchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radicek. 2023. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*. AAAI Press, 5131–5140. doi:10.1609/AAAI.V37I4.25642

[32] Svetoslav Karaivanov, Veselin Raychev, and Martin T. Vechev. [n. d.]. Phrase-Based Statistical Translation of Programming Languages. In *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*. 173–184.

[33] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.* 25, 3 (2020), 1980–2024.

[34] Marie-Anne Lachaux, Baptiste Rozière, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A Deobfuscation Pre-Training Objective for Programming Languages. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. 14967–14979.

[35] leetcode. [n. d.]. ([n. d.]). https://leetcode.cn/.

[36] Fang Liu, Jia Li, and Li Zhang. 2023. Syntax and Domain Aware Model for Unsupervised Program Translation. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 755–767. doi:10.1109/ICSE48619.2023.00072

[37] 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 Transactions on Software Engineering and Methodology* (2026).

[38] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

[39] Kaibo Liu, Zhenpeng Chen, Yiyang Liu, Jie M. Zhang, Mark Harman, Yudong Han, Yun Ma, Yihong Dong, Ge Li, and Gang Huang. 2025. LLM-Powered Test Case Generation for Detecting Bugs in Plausible Programs. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025*. 430–440.

[40] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*. IEEE, 456–467.

[41] Yizhou Liu, Pengfei Gao, Xinchen Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. 2024. MarsCode Agent: AI-native Automated Bug Fixing. *CoRR* abs/2409.00899 (2024). arXiv:2409.00899 doi:10.48550/ARXIV.2409.00899

[42] Llama. [n. d.]. ([n. d.]). https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct.

[43] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, and et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

[44] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. ACM, 101–114.

[45] minimumArrayLength. 2024.01. (2024.01). https://leetcode.cn/problems/minimize-length-of-array-using-operations/solutions/2613059/on-nao-jin-ji-zhuan-wan-pythonjavacgo-by-2lea.

[46] minOperations. 2024.03. (2024.03). https://leetcode.cn/problems/minimum-operations-to-exceed-threshold-value-ii/solutions/2664316/zui-xiao-dui-mo-ni-pythonjavacgo-by-endl-07jk/.

[47] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2013. Lexical statistical machine translation for language migration. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13,*. ACM, 651–654. doi:10.1145/2491411.2494584

[48] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*. IEEE Computer Society, 574–584. doi:10.1109/ASE.2015.36

[49] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, and et al. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 82:1–82:13.

[50] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, and et al. 2021. Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *CoRR* abs/2105.12655 (2021). arXiv:2105.12655

[51] Alfin Wijaya Rahardja, Junwei Liu, Weitong Chen, Zhenpeng Chen, and Yiling Lou. 2025. Can Agents Fix Agent Issues?. In *Advances in Neural Information Processing Systems 39: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025*.

[52] Daniel Ramos, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2024. BatFix: Repairing language model-based transpilation. *ACM Trans. Softw. Eng. Methodol.* 33, 6 (2024), 161. doi:10.1145/3658668

[53] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *CoRR* abs/2009.10297 (2020). arXiv:2009.10297

[54] Denise Rey and Markus Neuhäuser. 2011. Wilcoxon-Signed-Rank Test. In *International Encyclopedia of Statistical Science*, Miodrag Lovric (Ed.). Springer, 1658–1659. doi:10.1007/978-3-642-04898-2_616

[55] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

[56] Baptiste Rozière, Jie Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. [n. d.]. Leveraging Automated Unit Tests for Unsupervised Code Translation. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

[57] Sharpen. 2020. https://github.com/mono/sharpen.

[58] T. Sherwood, E. Perelman, and B. Calder. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. 3–14. doi:10.1109/PACT.2001.953283

[59] 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. *CoRR* abs/2412.14234 (2024). arXiv:2412.14234 doi:10.48550/ARXIV.2412.14234

[60] Marc Szafraniec, Baptiste Rozière, Hugh Leather, Patrick Labatut, François Charton, and Gabriel Synnaeve. 2023. Code Translation with Compiler Representations. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

[61] Qingxiao Tao, Tingrui Yu, Xiaodong Gu, and Beijun Shen. 2024. Unraveling the Potential of Large Language Models in Code Translation: How Far are We?. In *31st Asia-Pacific Software Engineering Conference, APSEC 2024, Chongqing, China, December 3-6, 2024*. IEEE, 353–362. doi:10.1109/APSEC65559.2024.00046

[62] TIOBE Software. 2025. TIOBE Programming Community Index. https://www.tiobe.com/tiobe-index

[63] TransAGENT. 2025. (2025). https://github.com/FudanSELab/TransAgent

[64] Bo Wang, Ruishi Li, Mingkai Li, and Prateek Saxena. 2023. TransMap: Pinpointing Mistakes in Neural Code Translation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. ACM, 999–1011.

[65] Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. 2025. Boosting Static Resource Leak Detection via LLM-based Resource-Oriented Intention Inference. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2905–2917. doi:10.1109/ICSE55347.2025.00131

[66] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

[67] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

[68] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. ACM, 172–184. doi:10.1145/3611643.3616271

[69] Yixi Wu, Pengfei He, Zehao Wang, Shaowei Wang, Yuan Tian, and Tse-Hsun Chen. 2024. A Comprehensive Framework for Evaluating API-oriented Code Generation in Large Language Models. *CoRR* abs/2409.15228 (2024). arXiv:2409.15228 doi:10.48550/ARXIV.2409.15228

[70] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents. *CoRR* abs/2407.01489 (2024). arXiv:2407.01489 doi:10.48550/ARXIV.2407.01489

[71] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. The Plastic Surgery Hypothesis in the Era of Large Language Models. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 522–534. doi:10.1109/ASE56229.2023.00047

[72] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1482–1494. doi:10.1109/ICSE48619.2023.00129

[73] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 959–971. doi:10.1145/3540250.3549101

[74] Yiqing Xie, Atharva Naik, Daniel Fried, and Carolyn P. Rosé. 2023. Data Augmentation for Code Translation with Comparable Corpora and Multiple References. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*. Association for Computational Linguistics, 13725–13739.

[75] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, and et al. 2018. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *CoRR* abs/1811.04211 (2018). arXiv:1811.04211

[76] Boyang Yang, Haoye Tian, Jiadong Ren, Shunfu Jin, Yang Liu, Feng Liu, and Bach Le. 2025. Enhancing Repository-Level Software Repair via Repository-Aware Knowledge Graphs. *CoRR* abs/2503.21710 (2025). arXiv:2503.21710 doi:10.48550/ARXIV.2503.21710

[77] Liming Yang, Junyu Luo, Xuanzhe Liu, Yiling Lou, and Zhenpeng Chen. 2026. BAMAS: Structuring Budget-Aware Multi-Agent Systems. In *Proceedings of the 40th Annual AAAI Conference on Artificial Intelligence, AAAI 2026*.

[78] 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 (2024), 1585–1608. doi:10.1145/3660778

[79] Zihan Yu, Liang He, Zhen Wu, Xinyu Dai, and Jiajun Chen. 2023. Towards Better Chain-of-Thought Prompting Strategies: A Survey. *CoRR* abs/2310.04959 (2023). arXiv:2310.04959 doi:10.48550/ARXIV.2310.04959

[80] Yuan Yuan and Wolfgang Banzhaf. 2020. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Trans. Software Eng.* 46, 10 (2020), 1040–1067. doi:10.1109/TSE.2018.2874648

[81] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2024. A Survey of Learning-based Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 2 (2024), 55:1–55:69. doi:10.1145/3631974

[82] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. Gamma: Revisiting Template-Based Automated Program Repair Via Mask Prediction. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 535–547.

[83] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*. ACM, 1592–1604. doi:10.1145/3650212.3680384

[84] Ming Zhu, Karthik Suresh, and Chandan K. Reddy. 2022. Multilingual Code Snippets Training for Program Translation. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*. AAAI Press, 11783–11790. doi:10.1609/AAAI.V36I10.21434

[85] Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. 2023. Tare: Type-Aware Neural Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*.

IEEE, 1443–1455. doi:10.1109/ICSE48619.2023.00126

Received 2025-09-10; accepted 2025-12-22