TRANSAGENT: An LLM-Based Multi-Agent System for Code Translation

Zhiqiang Yuan, Weitong Chen, Hanlin Wang, Kai Yu, Xin Peng, Yiling Lou
Department of Computer Science, Fudan University, China
{zhiqiangyuan23, wtchen21, 23210240303, 23262010042}@m.fudan.edu.cn
{pengxin, yilinglou}@fudan.edu.cn

***Abstract*—Code translation converts code from one programming language to another while maintaining its original functionality, which is crucial for software migration, system refactoring, and cross-platform development. Traditional rule-based methods rely on manually-written rules, which can be time-consuming and often result in less readable code. To overcome this, learning-based methods have been developed, leveraging parallel data to train models for automated code translation. More recently, the advance of Large Language Models (LLMs) further boosts learning-based code translation. Although promising, LLM-translated program still suffers from diverse quality issues (*e.g.,* syntax errors and semantic errors). In particular, it can be challenging for LLMs to self-debug these errors when simply provided with the corresponding error messages.**

In this work, we propose a novel LLM-based multi-agent system TRANSAGENT, which enhances LLM-based code translation by fixing the syntax errors and semantic errors with the synergy between four LLM-based agents, including *Initial Code Translator*, *Syntax Error Fixer*, *Code Aligner*, and *Semantic Error Fixer*. The main insight of TRANSAGENT is to first localize the error code block in the target program based on the execution alignment between the target and source program, which can narrow down the fixing space and thus lower down the fixing difficulties. To evaluate TRANSAGENT, we first construct a new benchmark from recent programming tasks to mitigate the potential data leakage issue. On our benchmark, TRANSAGENT outperforms the latest LLM-based code translation technique UniTrans in both translation effectiveness and efficiency; additionally, our evaluation on different LLMs show the generalization of TRANSAGENT and our ablation study shows the contribution of each agent.

I. INTRODUCTION

Code translation transforms program from one programming language to another while maintaining its original functionality. It is valuable for cross-language migration, enabling organizations to shift their code bases to newer languages for better performance [1], [2], [3]. It also helps modernize legacy systems by rewriting them in languages that improve maintainability and scalability during system refactoring [4], [1], [5]. Additionally, in large companies using multiple languages, code translation enhances interoperability and boosts programmer efficiency. Therefore, automated code translation techniques are essential for accelerating migration, reducing costs, and improving development efficiency.

Traditional rule-based code translation involves manually-written rules, which convert the source program into the program in the target language (*i.e.,* target program). However, such approaches require human experts to invest significant time and manual efforts in crafting rules, and the translated target program often suffers from poor readability and usability [6]. To address these issues, a series of learning-based code translation methods have been proposed to improve translation effectiveness [6], [7], [8]. These methods train models on large amounts of parallel data (*i.e.,* the pair of source and target program), allowing models to learn the translation patterns and mappings between different languages during training. However, high-quality parallel data for training is often scarce in practice [7], [9], [10], and the process of model training is also very time-consuming. For example, training the TransCoder model requires 32 V100 GPUs over 12 days [6].

The recent advance in Large Language Models (LLMs) have further boosts learning-based code translation.

However, Pan et al. [11] show that the target program generated by LLMs still suffers from various quality issues, such as compilation errors or functional discrepancies. To address these challenges, Yang et al. [12] propose UniTrans to enhance LLM-based code translation with an iterative fixing procedure. In particular, UniTrans leverages LLMs to fix the translated program based on the test inputs and outputs or compilation error messages. Although showing promise, UniTrans still fails to fix the translated program for a significant portion of cases, especially when built upon small LLMs (*e.g.,* with less than 10 billion parameters). For example, UniTrans with LLaMA-7B can only improve the translation accuracy from 31.25% to 31.90% (*i.e.,* only 0.65% improvement) for Java-to-Python translation. Therefore, improving the correctness of LLM-translated program still remains as a key challenge in this domain.

Typically, the errors in translated program can be divided into *syntax errors* and *semantic errors*. *Syntax errors* refer to the compilation errors or interpreting errors in the target program before its execution, while *semantic errors* occur when the target program exhibit different runtime behaviors (*i.e.,* different outputs) from the source program during execution. Syntax errors typically result from violating the target language grammars, which can be easily pinpointed by the syntax checkers or compilers. In contrast, semantic errors arise from functional discrepancies between the target program and the source program, which can be more challenging to fix, as they require LLMs to reason and to understand the execution of both source and target program. However, existing LLMs exhibit limited capabilities of reasoning the runtime behaviors during program execution [13], which explains why UniTrans has limited effectiveness in fixing errors with only test inputs/outputs provided.

This work. To enhance LLMs in code translation, we propose TRANSAGENT, an LLM-based multi-agent system that fixes both syntax errors and semantic errors in LLM-based code translation. *The main insight of* TRANSAGENT *is to first localize the error code block in the target program based on execution alignment, which can narrow down the fixing space and thus lower down the fixing difficulties.* TRANSAGENT includes four different LLM-based agents which can collaborate with each other, including *Initial Code Translator*, *Syntax Error Fixer*, *Code Aligner*, and *Semantic Error Fixer*. First, *Initial Code Translator* generates a set of tests based on the given source program, and then it generates an initial version of the target program via the basic code translation capability of the backbone LLM for the given source program and generated tests; Second, *Syntax Error Fixer* iteratively addresses the syntax errors in the target program based on compilation or interpreting error messages by first drafting a fixing plan and then generating the concrete patches; Third, *Code Aligner* divides the source program into blocks based on the control flow graph, and then leverages LLMs to map each block of the source program to that of the target program; Lastly, based on the mapped blocks between the source and target program, *Semantic Error Fixer* first localizes the error 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. We design TRANSAGENT in such a multi-agent system framework, as it can extend the standalone LLMs with the capabilities of acting to dynamic environments as well as the collaboration between different specialized agents. As summarized in the recent survey on LLM-based agents for software engineering [14], multi-agent systems have demonstrated promise in various software engineering tasks.

To evaluate TRANSAGENT, we first construct a new benchmark from recent programming tasks to mitigate the potential data leakage issue. On our benchmark, we first assess the overall translation effectiveness of TRANSAGENT, and the results show that it outperforms both the state-of-the-art LLM-based technique UniTrans [12] and the learning-based technique TransCoder [6]. We then conduct an ablation study to analyze the contribution of each agent in TRANSAGENT, as well as comparing them with those fixing strategies in UniTrans. The results show that both *Syntax Error Fixer* and *Semantic Error Fixer* in TRANSAGENT substantially enhance translation performance and both of them are more effective than the corresponding strategies in UniTrans. Additionally, to evaluate the mapping accuracy of TRANSAGENT, we conduct a user study to compare *Code Aligner* in TRANSAGENT with the existing code mapping approach TransMap [15] for code translation. The results show that our mapping agent *Code Aligner* (which is designed upon the synergy of both control-flow analysis and LLMs), substantially outperforms the existing mapping strategy TransMap (which is purely relied on LLMs) by a 39.6% improvement in mapping accuracy. Lastly, we evaluate the costs and generalization capabilities of TRANSAGENT. The results show that TRANSAGENT is more cost-effective than baselines, and can be generalized to different LLMs with consistent effectiveness.

In summary, this paper makes the following contributions:

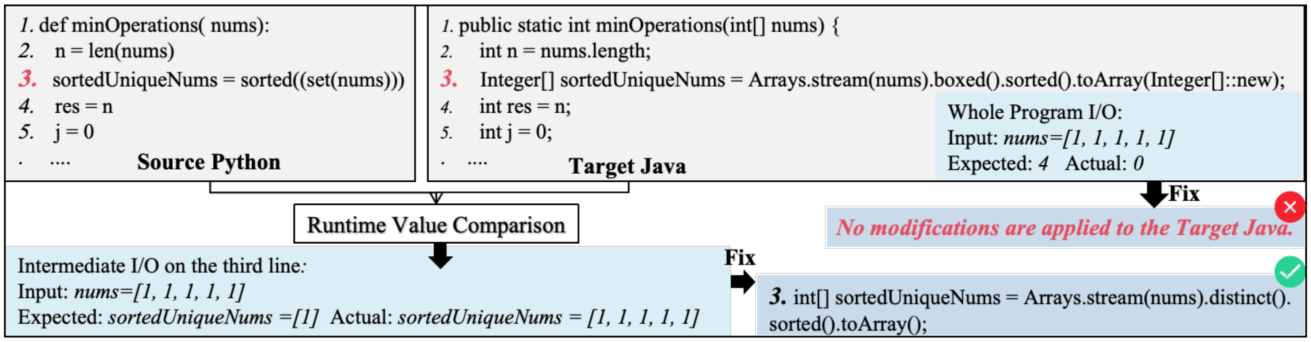
- **A Novel LLM-based Code Translation Technique.** We propose TRANSAGENT, an LLM-based multi-agent system for fixing the syntax and semantic errors in LLM-based code translation.
- **A Novel Code Mapping Strategy.** We design a code mapping strategy (*i.e., Code Aligner*) upon the synergy between control-flow analysis and LLMs.
- **A New Code Translation Benchmark.** We build a new code translation benchmark, which are 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, costs, generalization, and the ablation study of each agent.

II. MOTIVATING EXAMPLE

In this section, we illustrate the error-fixing challenges in the latest LLM-based code translation technique, UniTrans [12] via two examples. Here, we use the version of UniTrans built upon the backbone LLM Deepseek-coder-6.7b-instruct [16]. The first example is a failed case that UniTrans cannot successfully fix the *syntax error*, while the second example is a failed case that UniTrans cannot successfully fix the *semantic error*.

Challenges in fixing syntax errors sorely with the error messages thrown by the compiler/interpreter. When fix the the syntax errors, UniTrans uses the error messages provided by the compiler to fix the translation errors. However, these messages are often vague or lack specific repair guidance, making it difficult for the model to effectively resolve syntax errors. For example, Figure 2 shows a translated target Java program `minimumArrayLength` [17] from the source Python program, which encounters a compilation error: `no suitable method found for min(List<int[]>)`. The error message does not clarify why the method call fails or what needs to be changed, providing limited hints for the LLMs to generate correct patches. To enhance the ability of model to correct syntax errors, it is helpful to convert compiler error messages into clearer, more specific fix suggestions. For instance, in Figure 2, the error message can be rephrased to explain that `Collections.min()` cannot accept primitive types and requires a `List` of objects instead, which can potentially provide detailed guidance for LLMs in understanding and fixing the syntax error in the target program.

Challenges in fixing semantic errors sorely with whole program input and output. When fixing the semantic errors,



```
1. def minOperations( nums):
2.    n = len(nums)
3.    sortedUniqueNums = sorted((set(nums)))
4.    res = n
5.    j = 0
.     ....
```

```
1. public static int minOperations(int[] nums) {
2.    int n = nums.length;
3.    Integer[] sortedUniqueNums = Arrays.stream(nums).boxed().sorted().toArray(Integer[]::new);
4.    int res = n;
5.    int j = 0;
.     ....
```

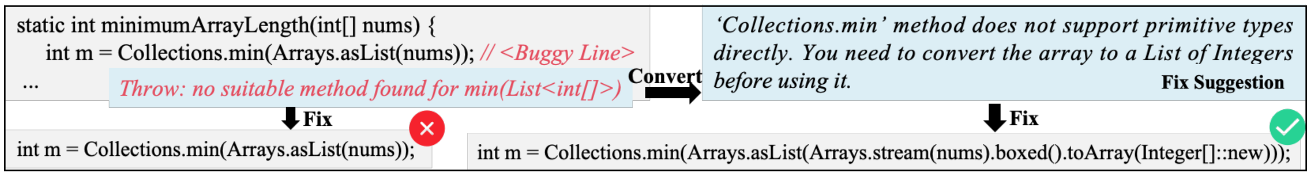
Whole Program I/O:
Input: *nums=[1, 1, 1, 1, 1]*
Expected: *4* Actual: *0*

No modifications are applied to the Target Java.

Intermediate I/O on the third line:
Input: *nums=[1, 1, 1, 1, 1]*
Expected: *sortedUniqueNums =[1]* Actual: *sortedUniqueNums = [1, 1, 1, 1, 1]*

```
3. int[] sortedUniqueNums = Arrays.stream(nums).distinct().
sorted().toArray();
```

Figure 1: Example of Fixing Semantic Errors in Target Java Program



```
static int minimumArrayLength(int[] nums) {
    int m = Collections.min(Arrays.asList(nums)); // <Buggy Line>
...
```

Throw: no suitable method found for min(List<int[]>)

'Collections.min' method does not support primitive types directly. You need to convert the array to a List of Integers before using it.

```
int m = Collections.min(Arrays.asList(nums));
```

```
int m = Collections.min(Arrays.asList(Arrays.stream(nums).boxed().toArray(Integer[]::new)));
```

Figure 2: Example of Fixing Syntax Errors in Target Java Program

UniTrans relies on the test inputs and outputs of the entire program to fix the target program. However, the inputs and outputs of the entire program can be too difficult for LLMs to utilize, as they require LLMs to reason along the execution paths of the entire program. As shown by recent study [13], LLMs exhibit limit capabilities in reasoning program execution, especially when there are multiple logical branches. Figure 1 illustrates the Java program called `minOperations` [18], which is translated from the Python program with UniTrans. The LLM fails to catch a key operation in the Line 3 of the source Python program (which actually deduplicates elements in `nums`). As a result, the target Java program does not perform this deduplication, thus returning an incorrect output of `0` instead of `4`. When UniTrans attempts to fix the semantic error with the test inputs and outputs, it is unsuccessful due to the subtle logical difference and the program complexity. To improve the model capabilities in fixing semantic error, it can be helpful to first decompose the problem by first localizing the fine-grained error location (*e.g.,* the error statement or block) and then fixing the error part only with runtime values related to it. For example, by comparing the runtime value of the variable between the source Python and target Java program, the error can be pinpointed to Line 3 of the Java program. Such a fine-grained error location, along with its relevant intermediate runtime values, can provide more detailed hints for LLMs to fix semantic errors.

III. Approach

Figure 3 illustrates the workflow of TransAgent. In particular, to effectively fix both syntax and semantic errors in LLM-based code translation, the main novelty of TransAgent is to first localize the error code block of the LLM-based translated target program by comparing the runtime behaviors between source program and target program, which can reduce the fixing space and thus lower down the fixing difficulties. To this end, TransAgent is designed as a multi-agent system with four different LLM-based agents which can collaborate with each other, including *Initial Code Translator*, *Syntax Error Fixer*, *Code Aligner*, and *Semantic Error Fixer*.

- **Initial Code Translator** first generates a set of tests based on the given source program; and then given the generated tests and the source program, it generates an initial version of the target program with the basic code translation capability of the backbone LLM. Note that *Initial Code Translator* is a basic component commonly used in previous work (*e.g.,* UniTrans [12]), which is not a contribution of TransAgent.
- **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. Different from how previous LLM-based code translation techniques [11], [12] fixing syntax errors, *the main novelty of Syntax Error Fixer in* TransAgent *is the planning of fix strategy.* In particular, based on error messages, *Syntax Error Fixer* first queries LLMs to generate a plan of the fixing strategy, based on which the concrete patches are further generated with 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 source program and target program in the following *Semantic Error Fixer* Component. Different from previous code mapping strategies [15] 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 block level.*
- **Semantic Error Fixer** first localizes the suspicious block in the target program which exhibits different runtime

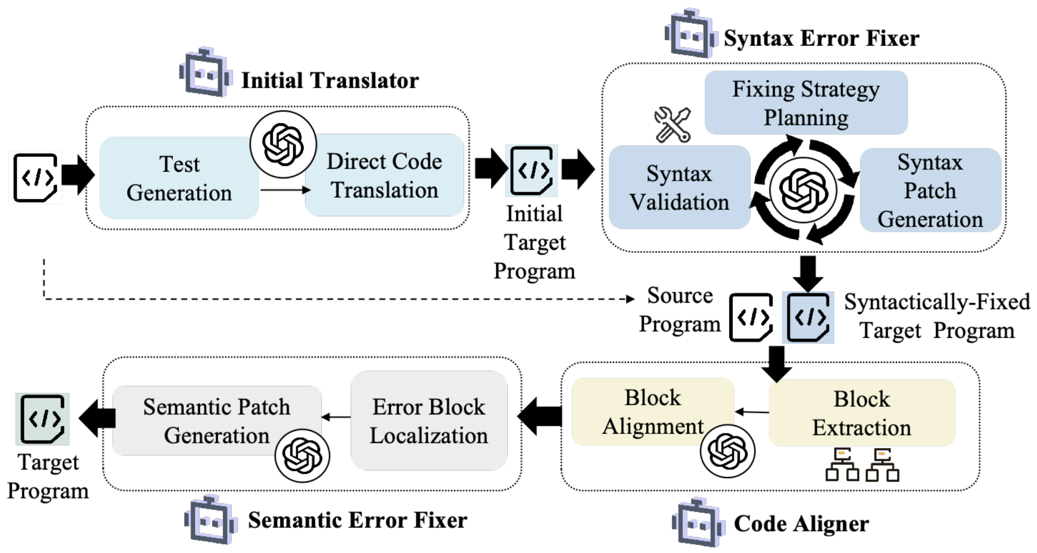
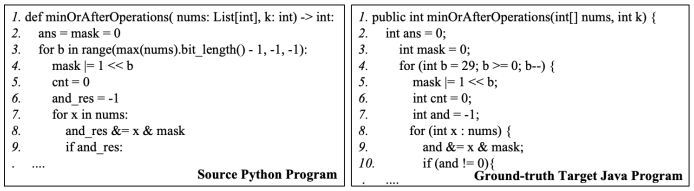


Figure 3: Workflow of TRANSAGENT

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 passing all the generated tests, the workflow terminates and the target program would be returned as the final target program; otherwise, the workflow proceeds to fix the syntax or semantic errors of the target program. In the following parts of this section, we use a Python-to-Java translation example `minOrAfterOperations` [19] (as shown in Figure 4) to illustrate the workflow of TRANSAGENT.



```
def minOrAfterOperations( nums: List[int], k: int) -> int:
    ans = mask = 0
    for b in range(max(nums).bit_length() - 1, -1, -1):
        mask |= 1 << b
        cnt = 0
        and_res = -1
        for x in nums:
            and_res &= x & mask
            if and_res:
    ....
```

Source Python Program

```
public int minOrAfterOperations(int[] nums, int k) {
    int ans = 0;
        int mask = 0;
        for (int b = 29; b >= 0; b--) {
            mask |= 1 << b;
            int cnt = 0;
            int and = -1;
           for (int x : nums) {
               and &= x & mask;
               if (and != 0){
    ....
```

Ground-truth Target Java Program

Figure 4: Source Python and Ground-truth Java Program of `minOrAfterOperations` [19]

A. Initial Code Translator

Following the previous code translation work UniTrans [12], *Initial Code Translator* mainly includes two parts, *i.e.,* test generation and direct code translation.

Test Generation. As revealed by UniTrans [12], 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 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.

In particular, as a basic component of LLM-based code translation, *Initial Code Translator* is not a contribution of TRANSAGENT and we mainly follow the prompt and settings from UniTrans [12]. The detailed prompt can be found in our replication package [20].

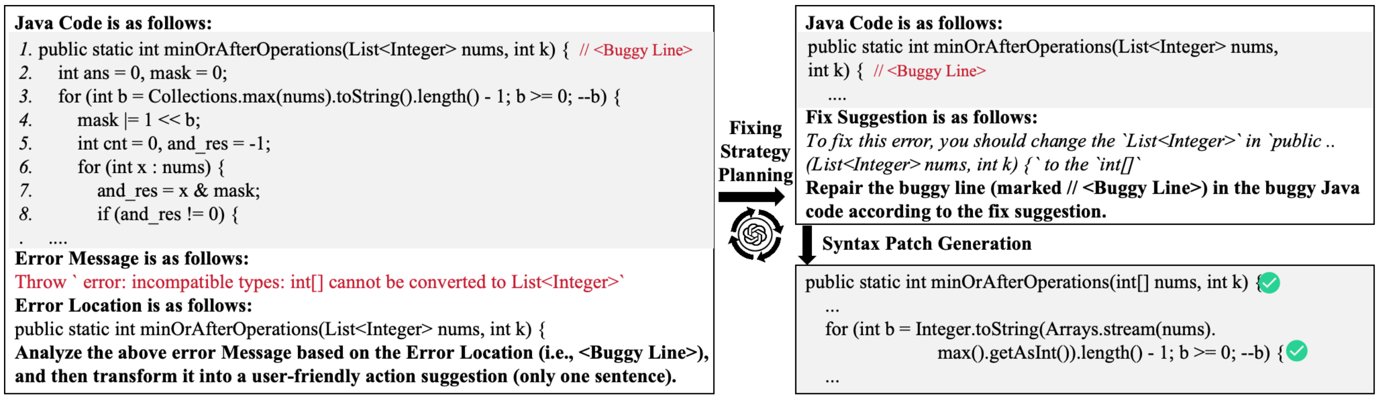
B. Syntax Error Fixer

Syntax Error Fixer iteratively leverages LLM to fix the syntax errors in the target program. In particular, it iteratively goes through three steps, *i.e.,* (i) *syntax validation*, (ii) *fixing strategy planning*, and (iii) *syntax patch generation*.

Syntax Validation. In this step, *Syntax Error Fixer* invokes external tools (*e.g.,* compilers for Java/C++ or interpreter for Python) to check the syntactic correctness of the target program. If no syntax errors are reported in this step, the target program would be passed to the next two agents (*i.e., Code Aligner* and *Semantic Error Fixer*); otherwise, *Syntax Error Fixer* proceeds to the following steps of *fixing strategy planning* and *patch generation*.

Fixing Strategy Planning. As mentioned in previous research [14], planning can further boost LLM-based agents for better effectiveness. Therefore, instead of directly leveraging LLMs to generate patches, *Syntax Error Fixer* first leverages LLMs to generate a plan of fixing strategies. As shown in Figure 5, LLMs are prompted to briefly describe the fixing strategy in natural language. The buggy location is determined by parsing the error message generated in the step of syntax validation.

Syntax Patch Generation. Based on the generated plan of fixing strategies, this step further prompts LLMs to generate concrete patches to fix the syntax error in the target program.



Java Code is as follows:

```
public static int minOrAfterOperations(List<Integer> nums, int k) {  // <Buggy Line>
    int ans = 0, mask = 0;
    for (int b = Collections.max(nums).toString().length() - 1; b >= 0; --b) {
        mask |= 1 << b;
        int cnt = 0, and_res = -1;
        for (int x : nums) {
            and_res = x & mask;
            if (and_res != 0) {
    ....
```

Error Message is as follows:

Throw ` error: incompatible types: int[] cannot be converted to List<Integer>`

Error Location is as follows:

```
public static int minOrAfterOperations(List<Integer> nums, int k) {
```

Analyze the above error Message based on the Error Location (i.e., <Buggy Line>), and then transform it into a user-friendly action suggestion (only one sentence).

Java Code is as follows:

```
public static int minOrAfterOperations(List<Integer> nums,
int k) {  // <Buggy Line>
    ....
```

Fix Suggestion is as follows:

To fix this error, you should change the `List<Integer>` in `public .. (List<Integer> nums, int k) {` to the `int[]`

Repair the buggy line (marked // <Buggy Line>) in the buggy Java code according to the fix suggestion.

```
public static int minOrAfterOperations(int[] nums, int k) {✓
    ...
    for (int b = Integer.toString(Arrays.stream(nums).
                        max().getAsInt()).length() - 1; b >= 0; --b) {✓
    ...
```

Figure 5: Prompts in *Syntax Error Fixer*

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) there are the same syntax errors occurring at the same buggy location as the previous iteration (to avoid being stuck in an endless loop). Otherwise, if there are syntax errors different from the previous iteration, TRANSAGENT continues the iterative fixing process.

C. Code Aligner

As shown by the motivating example in Section II, 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 source program and 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 source program and target program. The previous code mapping technique TransMap [15] 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 [15]. Therefore, to address these limitations, *Code Aligner* proposes a block-level mapping techniques, 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* which divides the source program into blocks via control-flow analysis, and (ii) *block alignment* which maps each block in source program to 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.

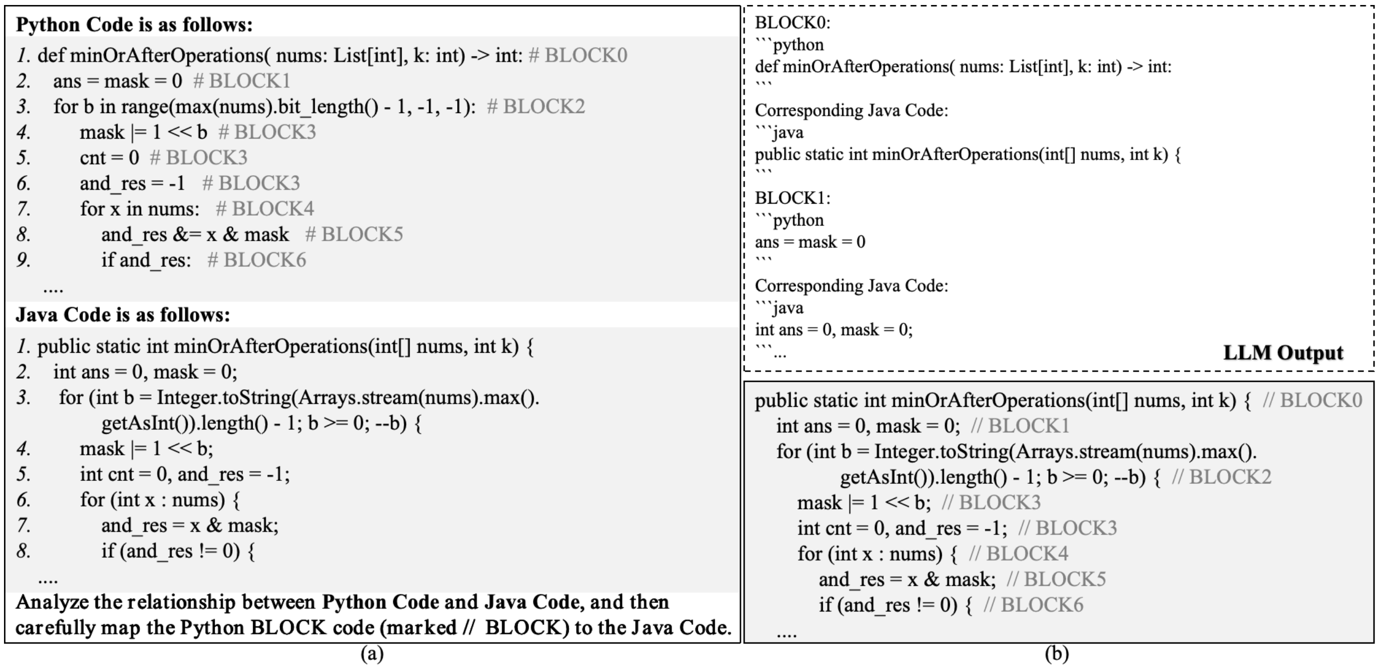
- A continuous sequence of statements that have no jumps in or out of the middle of a block would be regarded as a block. For example in Figure 6.a, Line 4 - 6 is a block (*i.e.,* marked as BLOCK3).
- Any control flow statement (*i.e.,* the statement that can result in different execution paths such as `while`, `for`, `try`, or `if`) would be regarded as a block. For example in Figure 6.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* [21] in control flow graph. However, the basic block is often on the granularity of 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 call the blocks in source program as source block and call the blocks in target program as target blocks. In this way, after the block extraction step, 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 6.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 6.a, LLMs are prompted to map the numbered source blocks to the corresponding target block; the top part of Figure 6.b shows the mapping outputs generated by LLMs, which are further post-processed into structured representation (as shown in the bottom part of Figure 6.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 the mapping function $f_{map}(B_{Si}) = B_{Tj}$.

D. 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



Python Code is as follows:

```
def minOrAfterOperations( nums: List[int], k: int) -> int: # BLOCK0
  ans = mask = 0  # BLOCK1
  for b in range(max(nums).bit_length() - 1, -1, -1):  # BLOCK2
      mask |= 1 << b  # BLOCK3
      cnt = 0  # BLOCK3
      and_res = -1  # BLOCK3
      for x in nums:  # BLOCK4
          and_res &= x & mask  # BLOCK5
          if and_res:  # BLOCK6
  ....
```

Java Code is as follows:

```
public static int minOrAfterOperations(int[] nums, int k) {
  int ans = 0, mask = 0;
   for (int b = Integer.toString(Arrays.stream(nums).max().
          getAsInt()).length() - 1; b >= 0; --b) {
      mask |= 1 << b;
      int cnt = 0, and_res = -1;
      for (int x : nums) {
          and_res = x & mask;
          if (and_res != 0) {
  ....
```

Analyze the relationship between Python Code and Java Code, and then carefully map the Python BLOCK code (marked // BLOCK) to the Java Code.

(a)

~~~
BLOCK0:
```python
def minOrAfterOperations( nums: List[int], k: int) -> int:
~~~
Corresponding Java Code:
```java
public static int minOrAfterOperations(int[] nums, int k) {
```
BLOCK1:
```python
ans = mask = 0
```
Corresponding Java Code:
~~~java
int ans = 0, mask = 0;
```...
~~~

LLM Output

```
public static int minOrAfterOperations(int[] nums, int k) {  // BLOCK0
   int ans = 0, mask = 0;  // BLOCK1
   for (int b = Integer.toString(Arrays.stream(nums).max().
            getAsInt()).length() - 1; b >= 0; --b) {  // BLOCK2
      mask |= 1 << b;  // BLOCK3
      int cnt = 0, and_res = -1;  // BLOCK3
      for (int x : nums) {  // BLOCK4
         and_res = x & mask;  // BLOCK5
         if (and_res != 0) {  // BLOCK6
   ....
```

(b)

Figure 6: Prompts in *Coder Aligner*

error information (*i.e., Semantic Patch Generation*). Different from previous LLM-based code translation work [12], [11] that directly leverages 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.

1) Error Block Localization: For error block localization, TRANSAGENT first collects the runtime values of blocks in both source and target program (*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 program. In particular, TRANSAGENT first instruments both source and target program by adding logging statements at the entry or exit of each block (more details are in Section IV-E); 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 program. 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 $V_T^{t_k} =< V_{T1}^{t_k}, V_{T2}^{t_k}, ..., V_{TL'}^{t_k} >$.

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 $B_{Te}$
1  for $t_k$ in $T$ do
2      $l \leftarrow 0$;
3      while $l < len(V_T^{t_k})$ do
4          $V_S \leftarrow V_S^{t_k}[l]$; $V_T \leftarrow V_T^{t_k}[l]$;
5          $B_{Si} \leftarrow f_{id}(V_S)$; $B_{Tj} \leftarrow f_{id}(V_T)$;
6          if $V_T == NULL$ then
7              return $f_{map}(B_{Si})$ ;
8          if $B_{Tj} \neq f_{map}(B_{Si})$ then
9              if $B_{Tj}$ is a control flow statement then
10                 return $B_{Tj}$ ;
11             else
12                 return $f_{map}(B_{Si})$ ;
13         else
14             if $Equal(V_S, V_T)$ then
15                 continue;
16             else
17                 return $B_{Tj}$ ;
18         $l \leftarrow l + 1$;
```

Runtime value comparison. As illustrated in Algorithm **??**, 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.

2) Semantic Patch Generation: After identifying the error block in 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 [13]. 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 V-B further confirm the complementarity between these two fixing strategies.

For example, Figure 7.a and Figure 7.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[22], [23], [24], [25]; in addition, both fixing strategies follow Chain-of-Thought reasoning prompts with two steps, which have been shown as effective in previous research [26], [27], [26], [28].

Example illustration. Figure 7.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 7.b illustrates the value-aware fixing strategy. After translated 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 into the prompt, the value-aware fixing strategy can remind LLMs of the potential type errors for leading 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.

IV. EXPERIMENTAL SETTING

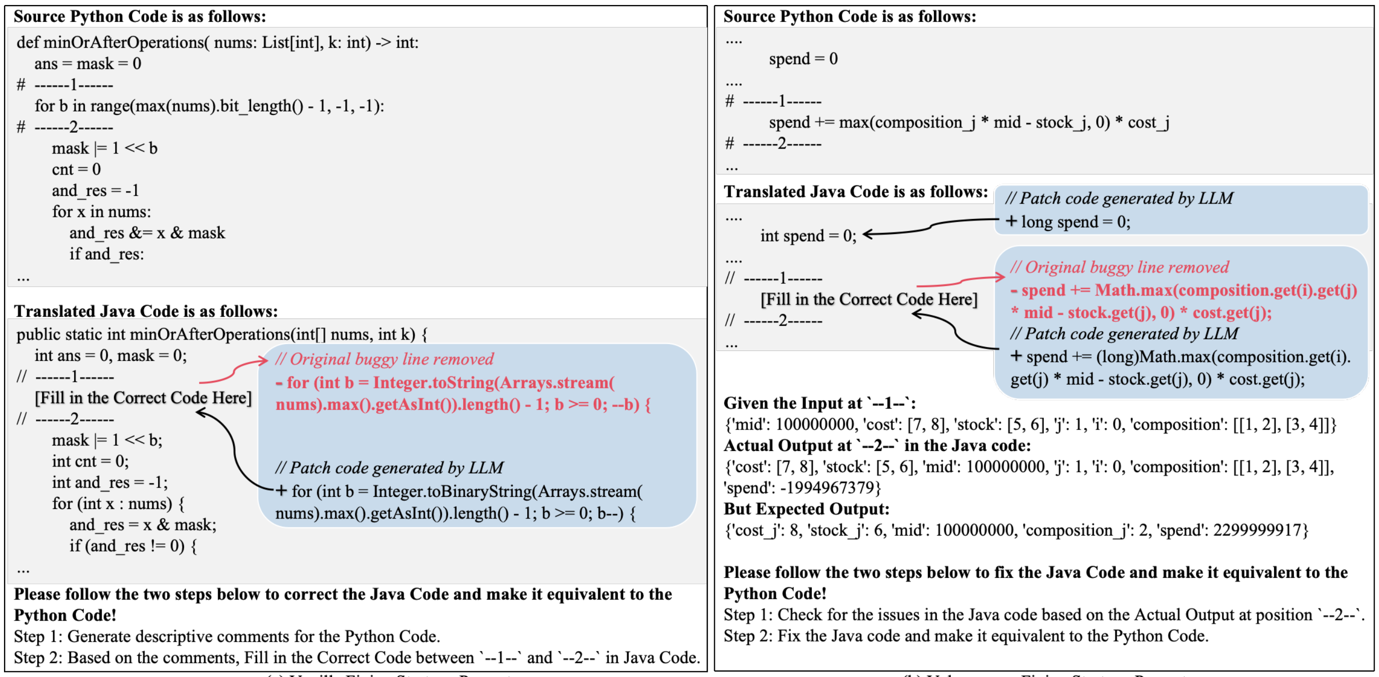
We evaluate TRANSAGENT by answering the following research questions.

- **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 (Mapping Accuracy):** How accurate is *Code Aligner* of TRANSAGENT in code mapping?
- **RQ4 (Cost):** How efficient is TRANSAGENT during code translation process?
- **RQ5 (Generalization):** How does TRANSAGENT perform with different backbone LLMs?

A. Benchmark

Limitations of Existing Benchmarks. Although there are many existing code translation benchmarks [6], [7], [29], [9], [10], they have the following limitations. First, all existing benchmarks suffer from potential data leakage issues, as their translation tasks are constructed from public programming competition by the training data timestamp of most recent LLMs (*e.g.,* the most widely-used evaluation dataset TransCoder-ST [6] is created in 2020). Second, some benchmarks involve the translation between only two languages [30], [31], [32], [33], such as the Java-C# benchmark CodeTrans [30] or the Java-Python benchmark AVATAR [31], which limits the generalization of evaluation. Third, some benchmarks (*e.g.,* CodeNet [34]) suffer from quality issues, and half of its tasks are manually identified as incorrect code by experts [35], thus can harm the soundness of the evaluation.

New Benchmark Construction. To address the limitations above (especially the data leakage issue), we first create a new benchmark for code translation, which is constructed on the recent programming tasks which are released after the training data timestamp of recent LLMs. In particular, from the programming competition websites (*e.g.,* LeetCode [36] and GeeksforGeeks [37]), we collect the solutions of programming



Source Python Code is as follows:

```
def minOrAfterOperations( nums: List[int], k: int) -> int:
    ans = mask = 0
#  ------1------
    for b in range(max(nums).bit_length() - 1, -1, -1):
#  ------2------
        mask |= 1 << b
        cnt = 0
        and_res = -1
        for x in nums:
            and_res &= x & mask
            if and_res:
...
```

Translated Java Code is as follows:

```
public static int minOrAfterOperations(int[] nums, int k) {
    int ans = 0, mask = 0;
//  ------1------
    [Fill in the Correct Code Here]
//  ------2------
        mask |= 1 << b;
        int cnt = 0;
        int and_res = -1;
        for (int x : nums) {
            and_res = x & mask;
            if (and_res != 0) {
...
```

```
// Original buggy line removed
- for (int b = Integer.toString(Arrays.stream(
nums).max().getAsInt()).length() - 1; b >= 0; --b) {

// Patch code generated by LLM
+ for (int b = Integer.toBinaryString(Arrays.stream(
nums).max().getAsInt()).length() - 1; b >= 0; b--) {
```

Please follow the two steps below to correct the Java Code and make it equivalent to the Python Code!
Step 1: Generate descriptive comments for the Python Code.
Step 2: Based on the comments, Fill in the Correct Code between `--1--` and `--2--` in Java Code.

(a).Vanilla Fixing Strategy Prompt

Source Python Code is as follows:

```
....
      spend = 0
....
#  ------1------
      spend += max(composition_j * mid - stock_j, 0) * cost_j
#  ------2------
...
```

Translated Java Code is as follows:

```
....
    int spend = 0;
....
//  ------1------
    [Fill in the Correct Code Here]
//  ------2------
...
```

```
// Patch code generated by LLM
+ long spend = 0;

// Original buggy line removed
- spend += Math.max(composition.get(i).get(j)
* mid - stock.get(j), 0) * cost.get(j);
// Patch code generated by LLM
+ spend += (long)Math.max(composition.get(i).
get(j) * mid - stock.get(j), 0) * cost.get(j);
```

Given the Input at `--1--`:
{'mid': 100000000, 'cost': [7, 8], 'stock': [5, 6], 'j': 1, 'i': 0, 'composition': [[1, 2], [3, 4]]}
Actual Output at `--2--` in the Java code:
{'cost': [7, 8], 'stock': [5, 6], 'mid': 100000000, 'j': 1, 'i': 0, 'composition': [[1, 2], [3, 4]], 'spend': -1994967379}
But Expected Output:
{'cost_j': 8, 'stock_j': 6, 'mid': 100000000, 'composition_j': 2, 'spend': 2299999917}

Please follow the two steps below to fix the Java Code and make it equivalent to the Python Code!
Step 1: Check for the issues in the Java code based on the Actual Output at position `--2--`.
Step 2: Fix the Java code and make it equivalent to the Python Code.

(b).Value-aware Fixing Strategy Prompt

Figure 7: Prompts in Semantic Patch Generation

tasks in different programming languages, which are released after *August 2023*. Specifically, we focus on three popular programming languages, *i.e.,* Java, Python, and C++. 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 [39] 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 I presents the line distribution of our benchmark; the average line coverage of each program with test cases achieve 98.4% for Python, 98.7% for Java, and 98.4% for C++, indicating the test sufficiency for each translation task.

Table I: Line Distribution of Our Benchmark

Line Distribution	Python (%)	Java (%)	C++ (%)
[0,5)	11.0	2.7	3.2
[5,10)	24.6	13.1	13.6
[10,15)	35.5	24.9	31.2
[15, 20)	16.2	26.7	20.8
[20,60]	12.7	32.6	31.2

B. Baselines

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

- **UniTrans [12]** is the latest LLM-based code translator which iteratively fixes translated program with LLMs. It is notable that another LLM-based technique proposed by Pan et al. [11] also shares the similar fixing approach as UniTrans, thus we do not include it as a separate baseline.
- **TransCoder [6]** is a representative learning-based code translation technique, which has been evaluated by almost all the previous code translation research [12], [7], [8], [9], [29], [10].

Code Mapping Baselines. In RQ3 of mapping accuracy evaluation, we compare *Code Aligner* in TRANSAGENT with [15], which is the latest LLM-based code mapping strategy for code translation.

C. Studied Models

In RQ5, we evaluate TRANSAGENT with the following LLMs to study the generalization of TRANSAGENT. In particular, we focus on models with fewer than 10 billion parameters given the resource constraints, and we only include the models whose training data cutoff is before our benchmark, to avoid data leakage issue.

- **Deepseek-coder-6.7b-instruct [16]** with 6.7 billion parameters, which is 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 [40]** 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 [41]** with 6 billion parameters, which is the second version of ChatGLM-6B [42] released in June 2023 and fine-tuned for general-purpose tasks.

D. Evaluation Metrics

Code Translation Metrics. Following previous work [7], [9], [6], we use the following metrics to evaluate the effectiveness of code translation techniques.

- **Computational Accuracy (CA)** [6], 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 program produce same outputs with the same test inputs.
- **CodeBLEU** [43], a metric for the similarity between target and source program.

Code Mapping Metrics. For RQ3, we calculate the mapping accuracy as $Accuracy = \#Correct\_Map$ $/\#Total\_Map$, where $\#Correct\_Map$ denotes the number of correct mappings between the source program and target program and $\#Total\_Map$ denotes the number of total mappings. In particular, a mapping is regarded as correct if all mapped target blocks and source blocks are semantically equivalent.

E. Implementation

Baseline Implementation. For UniTrans [12], we obtain its implementation from its replication package and make the following adjustment for comparison with TRANSAGENT. First, we replace the backbone LLM in UniTrans with the same LLM used in TRANSAGENT. In addition, we modify its fixing phase by splitting it into syntax error fixer and semantic error fixer, so as to compare with relevant components of TRANSAGENT. Furthermore, we change its fixed iteration strategy (*i.e.,* only iterating within a fixed threshold of iterations) into the same dynamic strategy as TRANSAGENT, for fair comparison. For TransCoder [6] 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 TransMap [15], we directly use its 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 [44], 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 an conversion helps standardize comparison across different programming languages, which might have varying data types and structures. In particular, we include the one-shot example in all prompts to guide LLMs to generate the required output format.

LLM Settings. For each studied open-source LLM, we use their released model and weights from HuggingFace [45]. To control randomness in our experiments, we set the parameters to "temperature=0" and "do_sample=False".

F. Experimental Procedure

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

RQ1 (Translation Effectiveness Evaluation). RQ1 compares the overall code translation effectiveness of TRANSAGENT with two baselines (*i.e.,* UniTrans [12] and TransCoder [6]) 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 [12] 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 ablation study. (i) *ICT*: including only *Initial Code Translator* for code translation; (ii) *ICT + SynEF*: including both *ICL* and *SynEF* agents; (iii) *ICT + SynEF + SemEF*: including *ICL*, *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 vanilla fixing strategy to the *ICT+SynEF+Val* variant.

RQ3 (Mapping Effectiveness). RQ3 compares the mapping accuracy of *Code Aligner* of TRANSAGENT with TransMap. We use both *Code Aligner* and TransMap to obtain mappings between target and source program. In particular, we then conduct a user study to manually inspect the mapping accuracy given the difficulties in automatically obtaining the mapping ground-truth. Given manual costs, we sample 290 pairs at a 95% confidence level with a 0.05 margin of error [46], [47]. Given the mutual translation among three languages, we select 48 data pairs from each pairwise translation. We invite two graduate students, who are not involved in our work but have rich experience in Python, Java, and C++, to independently annotate the mapping accuracy of TRANSAGENT and TransMap. Each student annotates the results of both TRANSAGENT and TransMap with the following annotation criteria: if all mapped code in the target and source program is semantically equivalent, the mapping is labeled as correct; otherwise, it is labeled as incorrect. A third student will provide an additional label to resolve disagreement with the majority voting strategy. In particular, we calculate the Cohen's kappa coefficient [48] for the label agreement. Lastly, we calculate the average accuracy of their annotations for each task, which serves as the final mapping accuracy for TRANSAGENT and TransMap.

RQ4 (Cost Evaluation). RQ4 collects the number of iterations and average time costs of TRANSAGENT. In particular, we also collect the translation accuracy improvement (CA) along each iteration.

RQ5 (Generalization Evaluation). RQ5 replace the default backbone LLM (*i.e.,* Deepseek-coder-6.7b-instruct) in TRANSAGENT with other two different LLMs (*i.e.,* Llama-3-8B-Instruct and ChatGLM2-6B).

V. EXPERIMENTAL RESULTS

A. *RQ1: Overall Effectiveness*

Table II shows the performance of TRANSAGENT and baselines. Overall, TRANSAGENT achieves the best performance across all six translation scenarios, especially in translations between dynamic and static languages. For example, in 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 summary, TRANSAGENT is effective by surpassing both learning-based and LLM-based baselines.

TransCoder performs the worst among studied transpilers in the six translation scenarios. Analysis of its output reveals that, while TransCoder generates complete target code, it contains numerous syntax and semantic errors. This issue arises because TransCoder, as a machine learning-based method, suffers from limited training data, leading to poor generalization when handling new data. In contrast, LLMs demonstrate stronger generalization due to their training on vast and diverse datasets. When comparing UniTrans and TRANSAGENT, their performance on CodeBLEU is quite similar, suggesting that the structures of the code they generate are comparable. However, TRANSAGENT outperforms UniTrans in CA, with an average performance improvement of 13.7%. This is because TRANSAGENT uses a more fine-grained approach to fix the errors in the target code. For instance, Figure 1 shows a case that UniTrans fails to correct an error, but TRANSAGENT succeeds by executing the target code, analyzing intermediate execution states, and pinpointing the exact location of the error, leading to more accurate code correction.

Translation Effectiveness Evaluation Summary: TRANSAGENT is an effective approach for code translation tasks, consistently outperforming existing state-of-the-art transpilers.

B. *RQ2: Ablation Evaluation*

Table III illustrates the contribution of each agent in UniTrans and TRANSAGENT to the overall translation performance. Overall, *Syntax Error Fixer* and *Semantic Error Fixer* in TRANSAGENT play a positive role in enhancing the performance of translations. For example, in the C++-to-Java task, these agents increase CA by 31.0% compared to *Initial Code Translator*, while in the Python-to-Java scenario, they achieve a 40.0% improvement. In summary, the error fixing agents employed in TRANSAGENT are effective in identifying and resolving errors in the target program, thereby enhancing the overall performance of code translation.

Both *Syntax Error Fixer* and *Semantic Error Fixer* in TRANSAGENT enhance performance in code translation tasks. For example, in the Python-to-Java translation scenario, *Syntax Error Fixer* improves the CA by 32.9%, while *Semantic Error Fixer* adds an additional 7.1%. Moreover, compared to *Syntax Error Fixer* and *Semantic Error Fixer* in UniTrans, these agents in TRANSAGENT are more effective in improving overall translation performance, especially when *Initial Code Translator* yields sub-optimal results. For instance, in the Python-to-Java scenario, *Initial Code Translator* achieves a CA of only 49.5%, with *Syntax Error Fixer* in UniTrans improving performance by just 5.7%, while *Syntax Error Fixer* in TRANSAGENT improves it by 32.9%. Similarly, *Semantic Error Fixer* in UniTrans increases CA by only 1.0%, whereas *Semantic Error Fixer* in TRANSAGENT boosts it by 7.1%. These results show that *Syntax Error Fixer* and *Semantic Error Fixer* in TRANSAGENT correct errors more effectively, outperforming the error correction strategies in UniTrans.

Both Value-aware and Vanilla fixing strategies in *Semantic Error Fixer* can improve the overall performance of TRANSAGENT in code translation tasks. 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: *Syntax Error Fixer* and *Semantic Error Fixer* agents in TRANSAGENT effectively enhance translation performance, and their repair strategies are more effective than those used in UniTrans.

C. *RQ3: Mapping Accuracy*

Table IV shows the performance of TransMap and *Code Aligner* in code mapping. The Cohen's Kappa value for the two annotators' labels is 0.811 for TransMap and 0.845 for *Code Aligner*, indicating a high level of agreement between the annotators. Overall, *Code Aligner* outperforms TransMap in mapping accuracy, with the most significant improvement in the C++-to-Python mapping task, showing a 39.6% increase. In the Python-to-C++ mapping task, the improvement is relatively lower but still reaches 6.2%. This indicates that the block-level mapping approach in *Code Aligner* achieves better alignment between source program and target program compared to the purely LLM-based mapping approach used in TransMap.

Error analysis reveals that purely LLM-based mapping of TransMap struggles when lines shift during translation, making accurate mapping difficult for smaller LLMs. For example, in Figure 8, when translating C++ code to Java, the second line

Table II: 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(%)	CodeBLEU	CA(%)	CodeBLEU	CA(%)	CodeBLEU	CA(%)	CodeBLEU	CA(%)	CodeBLEU	CA(%)	CodeBLEU
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**

Table III: 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**
Δ%	0.5	0.5	1.9	1.9	4.8	27.9	3.0	4.5	2.2	4.5	5.7	32.9
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**
Δ%	0.0	8.2	2.0	3.0	1.0	3.1	0.0	7.0	2.0	6.1	1.0	7.1
$\Delta_{\text{ICT+SynEF/Val}}$%	-	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

of C++ is translated into Java lines 2, 4, and 5, and the third line is translated into lines 4, 6, and 7. However, TransMap aligns lines sequentially, leading to mapping errors. In contrast, *Code Aligner* divides the source program into blocks based on the control flow before LLM-based mapping. For example, in Figure 8, lines 2, 3, and 4 of the C++ code are grouped as a block, which *Code Aligner* then maps directly to lines 2 through 8 of the Java code, resulting in correct mapping. This demonstrates that block-level mapping with the hybrid of control flow and LLMs is more robust for aligning source program with target program.

Mapping Accuracy Evaluation Summary: the hybrid mapping approach with both control flow analysis and LLMs in TransAgent substantially outperforms existing purely LLM-based code mapping method TransMap.

D. RQ4: Cost Evaluation

Table V shows the relationship between the number of iterations in TransAgent and its performance in translation tasks. TransAgent uses a dynamic iteration strategy, deciding whether to continue based on the outcomes of previous corrections rather than a fixed number of iterations. From the table V, we could observe that most iterations in TransAgent are completed within two rounds or less. For instance, in the Java-to-Python translation, TransAgent achieves a CA of 84.5% without iterations. After one and two iterations, the CA increases by 5.3% and 3.4%, with no further improvement beyond the second iteration. This indicates that the strong translation performance of TransAgent is not due to an excessive number of iterations.

Additionally, we conduct a time consumption analysis for TransAgent and UniTrans in translation tasks. The results show that TransAgent takes an average of 19 seconds per example, while UniTrans takes 24 seconds, indicating the better efficiency of TransAgent over UniTrans.

Cost Evaluation Evaluation Summary: TransAgent is more efficient than the baseline UniTrans in generating translated program of higher quality.

E. RQ5: Generalization Evaluation

Table VI presents the translation performance of each component in TransAgent across different LLMs. Overall, TransAgent demonstrates the ability to adapt to various LLMs, enhancing their performance in code translation tasks. For example, TransAgent improves performance by 31.5% (= 61.2%-29.7%) in the Python-to-Java task with Llama3 and by 38.5% (= 55.0%-16.5%) in the C++-to-Python task with ChatGLM. This demonstrates that the error correction strategies in TransAgent generalize well across different LLMs.

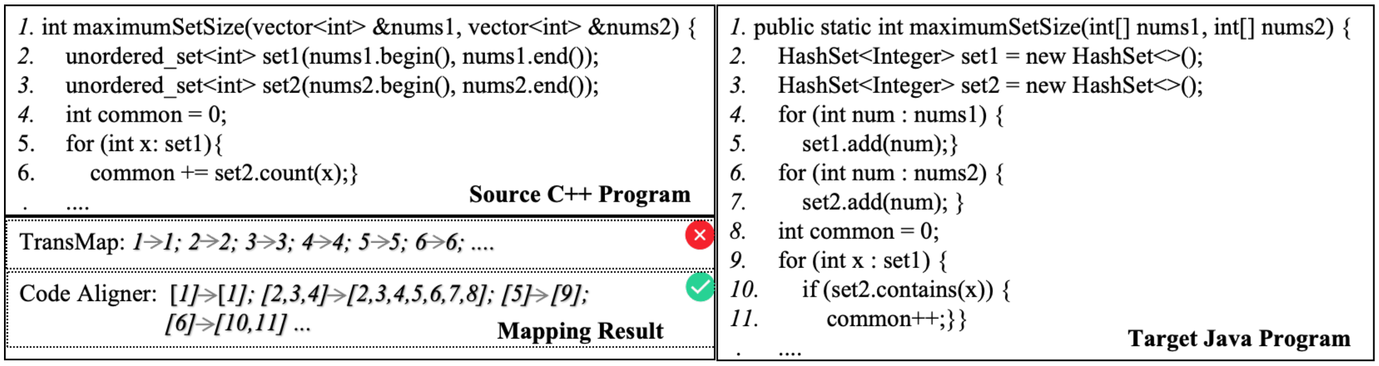
Both *Syntax Error Fixer* and *Semantic Error Fixer* show strong adaptability across LLMs, contributing to a stable improvement in translation performance. *Syntax Error Fixer* achieves the highest improvement of 25.3% with Llama3 in Python-to-Java and 6.9% with ChatGLM in Java-to-C++.

Similarly, *Semantic Error Fixer* shows the highest improvement of 9.2% in Llama3 for Java-to-Python and 38.0% in ChatGLM for C++-to-Python. This indicates that each component within TransAgent is capable of generalizing across different models, thereby enhancing the translation performance of TransAgent.

Generalization Evaluation Summary: TransAgent can be applied to different models to enhance their performance in code translation tasks.

VI. Threats to Validity

(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 LLM. 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 calculat line coverage to ensure the equivalence of the constructed source and target code. (iii) A further threat arises from the subjective judgments of annotators when evaluating code mapping performance. To resolve this, we invite



```
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

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

```
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

Figure 8: Example of Mapping Results of TransMap and TRANSAGENT

Table IV: Mapping Performance Comparison of TransMap and *Code Aligner* in TRANSAGENT

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

Table V: Percentage Improvement per Iteration for TRANSAGENT in CA

#Iteration	Java to Python	Java to C++	C++ to Java	C++ to Python	Python to C++	Python to Java
0	84.5%	89.6%	62.5%	84.0%	79.3%	50.0%
1	+5.3%	+4.0%	+26.5%	+10.0%	+6.1%	+36.2%
2	+3.4%	+0.5%	+1.0%	+0.5%	+2.0%	+1.4%
3+	+0.0%	+0.0%	+0.5%	+0.0%	0.0%	+2.0%

Table VI: Generalization of TRANSAGENT

CA (%)	Java to Python		Java to C++		C++ to Java		C++ to Python		Python to C++		Python to Java	
	Llma3	ChatGLM	Llma3	ChatGLM	Llma3	ChatGLM	Llma3	ChatGLM	Llma3	ChatGLM	Llma3	ChatGLM
ICT	72.9	18.4	75.1	7.5	53.5	6.0	79.5	16.5	66.5	4.5	29.7	5.4
ICT+SynEF	78.7	19.3	81.1	14.4	70.5	7.0	83.5	17.0	75.5	6.0	55.0	7.2
Δ%	5.8	0.9	6.0	6.9	17.0	1.0	4.0	0.5	9.0	1.5	25.3	1.8
IC+SynEF+SemEF	87.9	48.8	86.6	20.4	77.0	11.5	90.5	55.0	78.5	13.5	61.2	14.9
Δ%	9.2	29.5	5.5	6.0	6.5	4.5	7.0	38.0	3.0	7.5	6.2	7.7

two annotators independently to label the results and used Cohen's Kappa coefficient to assess the consistency of their labeled results. (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 [9] for implementing evaluation metrics like CodeBLEU.

VII. RELATED WORK

A. Code Translation

Early research on code translation proposes various approaches based on program analysis techniques. These methods apply manually formulated rules to carry out translations, such as C2Rust[49] and C2Go[50], which translate C program into Rust and Go, respectively, and Sharpen[51] and JavaSharp[52], which convert Java code into C#. However, rule creation is accomplished manually, which is time-consuming and limits the readability and accuracy of the target program [6]. To address the aforementioned issues, several learning-based code translation methods have emerged in recent years [32], [53], [54], [55]. Learning-based code translation methods have advanced significantly compared to rule-based approaches, but they still face a key challenge: the scarcity of parallel data for model training. To address this, researchers also propose various data augmentation techniques. For example, TransCoder-IR [8] uses compiler representations to link source and target program, MuST-PT [35] splits code into smaller equivalent chunks, and CMTrans [10] retrieves functionally similar code to expand datasets. Despite these efforts, data scarcity remains an issue, and the model training process is time-consuming.

LLMs have shown great potential in software engineering tasks like code generation [56], [57], [58], program repair [59], [60], and code summarization [61], [62]. Pan et al. [11] studied the performance of five LLMs (*e.g.,* Llama 2) on code translation and found challenges with syntax and semantic errors in their translations. Recently, Yang et al. introduced UniTrans [12], which uses test cases to improve LLM translation performance. However, UniTrans shows limited success in fixing translation errors, especially in models with fewer than 10 billion parameters, where improvements are minimal. To address this issue, we propose TRANSAGENT, a multi-agent translation system that leverages multiple agents to provide more detailed error correction information to enhance the code translation performance of LLMs.

B. Fault Location & Program Repair

Fault localization is crucial for automatic debugging and program repair. Existing fault localization techniques primarily include statistical analysis [63], coverage analysis [64], [65], machine learning [66], [67], [68], and LLM-based [69]. However, unlike existing fault localization methods designed for a single programming language, TRANSAGENT needs to identify errors by comparing code written in two different languages. In addition, existing fault localization techniques (such as spectrum-based fault localization) cannot be adapted to the code translation scenario, as most translated program exhibits very high coverage (*e.g.,*more than 98% line coverage) and it is almost infeasible for failed tests to distinguish the error block with coverage distribution. The closest method to TRANSAGENT is TransMap [15], which uses statement-level mapping to align source and target program. However, TransMap's reliance on line positions leads to errors when line order shifts during translation. To address this, TRANSAGENT introduces block-based mapping, using control flow analysis to map code blocks as the smallest unit. This approach is more robust, capturing logical and functional relationships better than line-based mapping, and significantly improving mapping accuracy. Additionally, TRANSAGENT can aid existing fault localization techniques. For example, providing better training data for learning-based methods [68], more precise reasoning for mutation-based techniques [70], and improved error tracking for tracing-based approaches [71].

For the program repair, early program repair efforts primarily relied on heuristic methods[72], [73], [74], constraint-based approaches[75], [76], [77], and pattern-based techniques[78], [79], [80]. However, these methods are heavily dependent on manually designed approaches or templates, leading to poor generalization capabilities in code repair. To address this issue, learning-based methods like SequenceR [81], CoCoNut [82], and Tare [83] have been proposed, achieving significant improvements by learning bug-fixing patterns from large code databases [84]. Recently, researchers have increasingly used LLMs for program repair, inspired by their success in various code-related tasks, and have seen better performance [60], [22], [85]. TRANSAGENT differs from existing LLM-based repair techniques [86] and RING [87], [22], [23], [24], [25] by proposing a novel fine-grained fixing paradigm, which further localizes the error block with aligned execution values.

VIII. CONCLUSION

In this paper, we introduce TRANSAGENT, a novel multi-agent system to improve LLM-based code translation with fine-grained execution alignment. On our benchmark, TRANSAGENT outperforms the latest LLM-based code translation technique UniTrans in both translation effectiveness and efficiency; additionally, our evaluation on different LLMs show the generalization of TRANSAGENT and our ablation study shows the contribution of each agent; lastly, we further perform a user study to find that the code mapping accuracy is substantially higher than the existing LLM-based code mapping strategy TransMap.

REFERENCES

[1] Sindre Grønstøl Haugeland, Phu Hong Nguyen, Hui Song, and Franck Chauvel. 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*, pages 170–177. IEEE, 2021.

[2] Transforming monolithic applications to microservices with mono2micro. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 3–3, 2021.

[3] Roberto Rodriguez Echeverria, Fernando Macias, Victor Manuel Pavon, Jose Maria Conejero, and Fernando Sanchez Figueroa. Legacy web application modernization by generating a rest service layer. *IEEE Latin America Transactions*, 13(7):2379–2383, 2015.

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

[5] Vikram Nitin, Shubhi Asthana, Baishakhi Ray, and Rahul Krishna. CARGO: ai-guided dependency analysis for migrating monolithic applications to microservices architecture. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 20:1–20:12. ACM, 2022.

[6] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 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*, 2020.

[7] Baptiste Rozière, Jie Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 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.

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

[9] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 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*, pages 1520–1534. Association for Computational Linguistics, 2023.

[10] Yiqing Xie, Atharva Naik, Daniel Fried, and Carolyn P. Rosé. 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*, pages 13725–13739. Association for Computational Linguistics, 2023.

[11] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, and et al. 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*, pages 82:1–82:13. ACM, 2024.

[12] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. Exploring and unleashing the power of large language models in automated code translation. *Proc. ACM Softw. Eng.*, 1(FSE):1585–1608, 2024.

[13] Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. Reasoning runtime behavior of a program with llm: How far are we? *arXiv e-prints*, 2024.

[14] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey, 2024.

[15] Bo Wang, Ruishi Li, Mingkai Li, and Prateek Saxena. 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*, pages 999–1011. ACM, 2023.

[16] deepseek-coder-6.7b instruct. 2023.

[17] minimumArrayLength. 2024.01.

[18] minOperations. 2024.03.

[19] minOrAfterOperations. 2024.01.

[20] TransAGENT. 2024.

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

[22] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 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*, pages 535–547. IEEE, 2023.

[23] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 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*, pages 172–184. ACM, 2023.

[24] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 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*, pages 522–534. IEEE, 2023.

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

[26] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 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*, 2022.

[27] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 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, 2023.

[28] Zihan Yu, Liang He, Zhen Wu, Xinyu Dai, and Jiajun Chen. Towards better chain-of-thought prompting strategies: A survey. *CoRR*, abs/2310.04959, 2023.

[29] Marie-Anne Lachaux, Baptiste Rozière, Marc Szafraniec, and Guillaume Lample. 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*, pages 14967–14979, 2021.

[30] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, and et al. 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*, 2021.

[31] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 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*, pages 2268–2281. Association for Computational Linguistics, 2023.

[32] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 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, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 651–654. ACM, 2013.

[33] Xinyun Chen, Chang Liu, and Dawn Song. 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, 2018.

[34] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, and et al. Project codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. *CoRR*, abs/2105.12655, 2021.

[35] Ming Zhu, Karthik Suresh, and Chandan K. Reddy. 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*, pages 11783–11790. AAAI Press, 2022.

[36] leetcode.

[37] geeksforgeeks.

[38] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *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*, 2023.

[39] gpt-4o mini. 2024.

[40] Llama-3-8B-Instruct. 2023.

[41] ChatGLM2-6B. 2023.

[42] ChatGLM-6B. 2023.

[43] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020.

[44] joern.

[45] huggingface.

[46] Ravindra Singh and Naurang Singh Mangat. *Elements of survey sampling*, volume 15. Springer Science & Business Media, 2013.

[47] Dennis Wackerly, William Mendenhall, and Richard L Scheaffer. *Mathematical statistics with applications*. Cengage Learning, 2014.

[48] J. Richard Landis and Gary G. Koch. An application of hierarchical kappa-type statistics in the assessment of majority agreement among multiple observers. *Biometrics*, 33(2):363–374, 1977.

[49] C2Rust. 2024.

[50] cxgo: C to Go transpiler. 2024.

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

[52] JavaToCSharp. https://github.com/paulirwin/JavaToCSharp, 2024.

[53] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Migrating code with statistical machine translation. In *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, pages 544–547. ACM, 2014.

[54] Svetoslav Karaivanov, Veselin Raychev, and Martin T. Vechev. 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*, pages 173–184. ACM, 2014.

[55] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 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, Lincoln, NE, USA, November 9-13, 2015*, pages 574–584. IEEE Computer Society, 2015.

[56] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, and et al. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[57] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *CoRR*, abs/2304.07590, 2023.

[58] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and improving chatgpt for unit test generation. *Proc. ACM Softw. Eng.*, 1(FSE):1703–1726, 2024.

[59] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 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*, pages 1469–1481. IEEE, 2023.

[60] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 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*, pages 1482–1494. IEEE, 2023.

[61] Toufique Ahmed and Premkumar T. Devanbu. 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*, pages 177:1–177:5. ACM, 2022.

[62] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. An empirical study on using large language models for multi-intent comment generation. *CoRR*, abs/2304.11384, 2023.

[63] Jinhan Kim, Gabin An, Robert Feldt, and Shin Yoo. Ahead of time mutation based fault localisation using statistical inference. In *32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, October 25-28, 2021*, pages 253–263. IEEE, 2021.

[64] Mike Papadakis and Yves Le Traon. Metallaxis-fl: mutation-based fault localization. *Softw. Test. Verification Reliab.*, 25(5-7):605–628, 2015.

[65] Yonghao Wu, Zheng Li, Yong Liu, and Xiang Chen. FATOC: bug isolation based multi-fault localization by using OPTICS clustering. *J. Comput. Sci. Technol.*, 35(5):979–998, 2020.

[66] Amr Mansour Mohsen, Hesham A. Hassan, Khaled Wassif, Ramadan Moawad, and Soha Makady. Enhancing bug localization using phase-based approach. *IEEE Access*, 11:35901–35913, 2023.

[67] Agnieszka Ciborowska and Kostadin Damevski. Fast changeset-based bug localization with BERT. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 946–957. ACM, 2022.

[68] Ziye Zhu, Yu Wang, and Yun Li. Trobo: A novel deep transfer model for enhancing cross-project bug localization. In *Knowledge Science, Engineering and Management - 14th International Conference, KSEM 2021, Tokyo, Japan, August 14-16, 2021, Proceedings, Part I*, volume 12815 of *Lecture Notes in Computer Science*, pages 529–541. Springer, 2021.

[69] Sungmin Kang, Gabin An, and Shin Yoo. A preliminary evaluation of llm-based fault localization. *CoRR*, abs/2308.05487, 2023.

[70] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning dynamic slices with confidence. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 169–180. ACM, 2006.

[71] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. REPT: reverse debugging of failures in deployed software. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 17–32. USENIX Association, 2018.

[72] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 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 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 298–309. ACM, 2018.

[73] Yuan Yuan and Wolfgang Banzhaf. ARJA: automated repair of java programs via multi-objective genetic programming. *IEEE Trans. Software Eng.*, 46(10):1040–1067, 2020.

[74] Matias Martinez and Martin Monperrus. ASTOR: a program repair library for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 441–444. ACM, 2016.

[75] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 416–426. IEEE / ACM, 2017.

[76] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, and et al. Nopol: Automatic repair of conditional statement bugs in java programs. *CoRR*, abs/1811.04211, 2018.

[77] Matias Martinez and Martin Monperrus. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings*, volume 11036 of *Lecture Notes in Computer Science*, pages 65–86. Springer, 2018.

[78] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. Tbar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 31–42. ACM, 2019.

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

[80] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 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*, pages 456–467. IEEE, 2019.

[81] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Trans. Software Eng.*, 47(9):1943–1959, 2021.

[82] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 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*, pages 101–114. ACM, 2020.

[83] Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. Tare: Type-aware neural program repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1443–1455. IEEE, 2023.

[84] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. A survey of learning-based automated program repair. *ACM Trans. Softw. Eng. Methodol.*, 33(2):55:1–55:69, 2024.

[85] Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. Pre-trained model-based automated software vulnerability repair: How far are we? *IEEE Trans. Dependable Secur. Comput.*, 21(4):2507–2525, 2024.

[86] Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, and Aseem Rastogi. Fixing rust compilation errors using llms. *CoRR*, abs/2308.05177, 2023.

[87] Harshit Joshi, José Pablo Cambronero Sánchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radicek. 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*, pages 5131–5140. AAAI Press, 2023.