

Enhancing Large Language Models in Long Code Translation through Instrumentation and Program State Alignment

XIN-YE LI, YA-LI DU, and MING LI, National Key Laboratory for Novel Software Technology, Nanjing University, China and School of Artificial Intelligence, Nanjing University, China

Code translation aims to transform code from one programming language to another while ensuring functional equivalence. It has broad applications in cross-platform development and software migration, such as transitioning C++-written systems to Rust. Recently, code translation has been greatly enhanced by the profound advancements in Large Language Models (LLMs). However, despite these improvements, existing LLM-based approaches struggle to infer a program's functionality from its appearance, leading to failure in preserving the program semantics during translation. This challenge becomes even more pronounced when dealing with longer and more complex code: current code LLMs face difficulties in handling lengthy programs and have limited capability in understanding intricate semantics. This limitation is particularly critical, as translating long code is more practical in real-world scenarios. To evaluate the capability of recent code LLMs on long code translation, we introduce *LongTrans*, a large-scale execution-based code translation benchmark. *LongTrans* consists of C++, Java, and Python programs, with lengths ranging from hundreds to thousands of tokens. Our empirical study on 12 series of LLMs reveals a dramatic performance decline as code length increases, regardless of model size. Even the most performant LLM, GPT-4o, achieves only 57.51% computational accuracy, highlighting the pressing need for further research in long code translation.

In this paper, we argue that the essence of code translation is maintaining invariant functionality while transforming its appearance (e.g. syntax, keywords) from one programming language to another. Although the translated code differs in appearance from the original, its program states should remain identical to the original from entry to exit throughout execution. To achieve this goal, we propose aligning program states at runtime beyond the final output to ensure functional consistency, ultimately producing an accurate translation. We propose an approach termed **Program State Alignment augmented Translation (PAST)**, to augment code LLMs with instrumentation, a classical technique in dynamic program analysis, to capture and describe the program states and assist their alignment during translation. Specifically, it is the first attempt to utilize code LLM to insert instrumentation code into both the original and the translated code and trace their program states at runtime. We prompt the code LLM to identify and repair the error based on the output traces, mitigating the inconsistency and thereby improving the accuracy of the translation. Extensive experimental results demonstrate that our approach significantly improves the performance of code LLMs in long code translation, with the computational accuracy increasing from 57.51% to 84.70% for GPT-4o, increasing from 50.68% to 69.97% for Mistral-Large-2, and increasing from 52.45% to 76.43% for DeepSeek-Coder-V2. Moreover, the results also show that these substantial improvements are consistent across different code LLMs and translation datasets, such as C++ to Java. Ablation studies further confirm that these improvements are brought by incorporating instrumentation and program state alignment.

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

Additional Key Words and Phrases: code translation, large language model

Authors' address: Xin-Ye Li, lixy@lamda.nju.edu.cn; Ya-Li Du, duyl@lamda.nju.edu.cn; Ming Li, lim@lamda.nju.edu.cn, National Key Laboratory for Novel Software Technology, Nanjing University, China and School of Artificial Intelligence, Nanjing University, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

0004-5411/2024/8-ART111 \$15.00

<https://doi.org/XXXXXX.XXXXXXX>

ACM Reference Format:

Xin-Ye Li, Ya-Li Du, and Ming Li. 2024. Enhancing Large Language Models in Long Code Translation through Instrumentation and Program State Alignment. *J. ACM* 37, 4, Article 111 (August 2024), 25 pages. <https://doi.org/XXXXXX.XXXXXXX>

1 INTRODUCTION

Code translation aims to transform programs from one language to another while preserving their original functionality. It effectively facilitates cross-language migration and enables organizations to modernize legacy systems for better performance, maintainability, and scalability [24, 33, 38, 47]. As software has evolved, various programming languages have been developed to address diverse needs, such as desktop, web, and mobile applications. This growing diversity has made it increasingly necessary to port software between languages to support expanding business platforms [5, 20, 68]. To meet this demand, different source-to-source translators, or compilers, have emerged over the past few decades, improving migration efficiency and enhancing interoperability across large companies using multiple languages [32, 48, 54, 55]. Automated code translation techniques, thus, remain essential in accelerating migration, reducing costs, and boosting development efficiency.

In recent years, learning-based approaches have achieved impressive performance on code-related tasks, such as code summarization [2, 8, 18, 22], code generation [11, 27, 43], bug localization [15, 45], and clone detection [65, 73]. Additionally, learning-based methods also have been proposed to enhance the efficiency and effectiveness of code translation [10, 14, 54, 55, 58], typically leveraging task-specific pre-training on large monolingual corpora. Although these approaches have led to notable improvements, their current performance remains insufficient for practical deployment. Recent advances in Large Language Models, pre-trained on billions of text and code tokens, present an alternative solution by eliminating the need for re-training or fine-tuning while demonstrating strong capability across various code-related tasks [9, 17, 19, 36, 39, 71]. Furthermore, recent studies have explored the performance of LLMs of code on code translation, showing promising results [31, 59, 72].

However, existing LLM-based approaches are translating code by appearance instead of inferring the underlying functionality and transforming the appearance from one programming language to another. Thus these approaches face challenges in preserving the program semantics during translation. Moreover, existing code translation benchmarks are typically limited to short code, such as snippet-level or function-level code, restricting their applicability in practice. In contrast, translating long code – at the program or repository level – is more complex and more common in real-world scenarios [52]. LLMs of code struggle to capture complex functionality from the appearance of long code inputs, even with extended context windows. Additionally, they are more prone to errors when generating long code outputs. Addressing these challenges is critical given the dominance of long code in practical software development.

To assess the capability of recent LLMs of code on long code translation scenarios, we conduct an empirical study on 12 series of LLMs, including GPT-4o [1], Llama-3.1 [62], Mistral-Large-2 [29], and DeepSeek-Coder [78], across six translation tasks, i.e. C++ to Python, Python to C++, Java to C++, C++ to Java, Java to Python and Python to Java. First, we construct an execution-based code translation benchmark called *LongTrans*, which consists of a large collection of programs in C++, Java, and Python. Based on *LongTrans*, we evaluate the above models under the metric of Computational Accuracy (CA). The results reveal a significant performance drop as the length of input code increases, with even the most capable LLM, GPT-4o, achieving only 57.51% computational accuracy, which is far from satisfactory. To better understand the causes of translation errors, we further analyze the failed samples in the translation results. Interestingly, whether due to a runtime error or a wrong answer, most failed samples tend to fail across all tests. This finding suggests

that the translation errors produced by LLMs of code are relatively superficial, reinforcing our assumption that LLMs of code have difficulty inferring the program's underlying functionality from its appearance. Recent work [72] attempts to repair the translation errors solely based on the execution feedback, such as error messages. However, identifying and repairing these errors in long code, which can span hundreds and thousands of lines, remains significant for current LLM-based approaches, akin to finding a needle in a haystack.

To address this challenge, we propose that the core of code translation lies in preserving the program's functionality and runtime behavior while transforming its appearance from one programming language to another. Although the program's appearance changes during translation, the translated program's states should remain consistent with the original from entry to exit throughout execution. To achieve this consistency, we propose a new approach termed **Program State Alignment augmented Translation (PAST)** to enhance the LLMs of code in long code translation through instrumentation and program state alignment. Our approach improves code translation by leveraging three key stages: Instrumentation, Translation, and Program State Alignment. In the Instrumentation stage, we insert the instrumentation code into the original program to monitor its program states and track the execution flow. During the Translation stage, LLMs of code translate the original program while preserving the inserted instrumentation. If the translated program fails to maintain consistency, we proceed to the Program State Alignment stage. In this final stage, we execute both the original and translated programs on public test inputs and address any detected misalignments through repair. We incorporate two repair strategies: direct repair and localize-and-re-translate, effectively enabling our approach to align the program states thus achieving correct translation.

We highlight our contributions in four key aspects:

- (1) We propose that the core of code translation is to maintain the original program's functionality while transforming the appearance from one programming language to another. This consistency is essential, as existing LLM-based methods struggle to infer the underlying functionality solely from code appearance.
- (2) We highlight the importance of long code translation and introduce *LongTrans*, a large-scale execution-based code translation benchmark containing C++, Java, and Python programs of varying complexities and lengths. Moreover, we conducted a comprehensive empirical study on 12 series of LLMs and carefully analyzed the results. Experimental findings reveal a significant decline in performance among LLMs of code as input code length increases, regardless of model size or programming language.
- (3) We propose a novel approach named **PAST** that utilizes instrumentation to capture the program states and trace the execution flow. By leveraging instrumentation, LLMs of code can effectively identify and address inconsistencies between the program states of both the original and the translated program, resulting in improved translation accuracy.
- (4) Experimental results demonstrate that our approach substantially enhances LLMs of code in long code translation, with the computational accuracy increasing from 57.51% to 84.70% for GPT-4o, increasing from 50.68% to 69.97% for Mistral-Large-2, and increasing from 52.45% to 76.43% for DeepSeek-Coder-V2. Comprehensive evaluation further confirms that our proposed approach's effectiveness is consistent across code LLMs and translation datasets.

2 BACKGROUND AND RELATED WORK

2.1 Automatic Code Translation

The automated code translation tool aims to construct a function $f(x)$, that maps source code x to target code $y = f(x)$. Early approaches to code translation were primarily rule-based, applying

techniques such as phrase-based statistical machine translation for program translation [5, 32, 46–48, 50], which leveraged the grammatical structures of programming languages for code migration.

In recent years, various deep learning techniques have been employed for program translation [14, 35, 66, 67]. Depending on whether the training data is monolingual or bilingual, these approaches can be categorized into supervised code translation [13, 77] and unsupervised/weakly supervised translation [54, 56, 58]. The latter focuses on using runtime testing with test cases, which is more valuable in practical development. Additionally, with the rise of large language models (LLMs) [72], code translation based on direct inference from LLMs has garnered widespread attention. Despite their remarkable achievements, these models still have clear limitations: (1) translating long code is challenging; (2) the time and space overhead for executing long code is large, leading to high evaluation costs; and (3) it is difficult to localize and repair defect when errors occur in execution accurately, especially for long code.

2.2 Large Language Models for Code

LLMs have demonstrated impressive performance on a variety of natural language processing (NLP) tasks [1] and have also attracted broad research interest. These LLMs are pre-trained on trillions of tokens, which often include a significant portion of code tokens sourced from open-source code-hosting platforms and developer communities, such as GitHub¹ and StackOverflow². As a result, many LLMs exhibit non-trivial code understanding and generation capability even though they are not specifically trained for code.

Inspired by the remarkable success of LLMs in natural language processing, researchers have turned their attention to enhancing the coding capabilities of LLMs. They discovered that continual pre-training on code significantly improves LLMs' performance on code-related tasks. Chen et al. [9] proposed Codex, a series of GPT-3 checkpoints pre-trained on 100B additional code tokens. Codex demonstrated a substantial improvement in code generation, solving 28.8% of hand-crafted programming problems whereas GPT-3 solved 0%. Moreover, existing works suggest that the performance of code generation can be further improved through self-consistency [64], candidate ranking [26, 44, 74], and other advanced prompting strategies. Besides code generation, LLMs also show great potential in other tasks, such as program repair [30], vulnerability detection [57], and code review [42].

3 MOTIVATION

This section presents the motivation behind our proposed approach. Our motivation is derived from a comprehensive empirical study conducted on our *LongTrans* benchmark. In this study, we assess the performance of recent code LLMs with varying parameter sizes across six translation datasets involving the mutual translation between C++, Java, and Python. The evaluation of translated programs is based on execution results, with the details described in Section 6.2. Therefore, evaluation metrics are execution-based, focusing on the compilation success rate and pass rate.

As illustrated in Figure 1, both compilation success rate and pass rate decline significantly as the number of input tokens increases. For smaller code LLMs, such as CodeLlama-7B [53], the pass rate drops to nearly zero when the input code exceeds 1024 tokens. Even more capable models, despite being fine-tuned for longer sequences, experience a 50% degradation in performance as the input code approaches 4096 tokens. This phenomenon suggests that, although code LLMs perform impressively on existing code translation benchmarks, they continue to face challenges in long code translation. The challenge in long code translation stems from two key factors: code LLMs

¹<https://github.com/>

²<https://stackoverflow.com/>

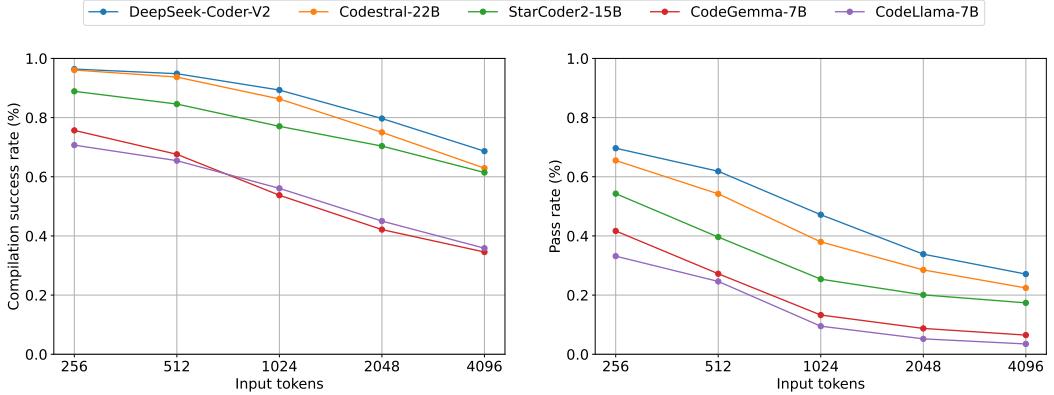


Fig. 1. Compilation success rate and pass rate drop dramatically as input length increases.

struggle to comprehend the full functionality of longer code inputs; second, they are more prone to mistakes when generating long code outputs. This issue persists across models of different sizes, indicating that long code translation is a common challenge for current code LLMs.

Take the most capable open-source code LLM (i.e., DeepSeek-Coder-V2 [78]) as an example and delve into failed samples derived from DeepSeek-Coder-V2's translation results. Labeling failed samples with detailed causes is labor-intensive and impractical with large-scale analysis. Therefore, we leverage the extensive tests in our *LongTrans* benchmark to estimate the quality and identify key characteristics of the failed samples. We execute all the translated programs on relevant tests and record their results. The failed samples are classified into five categories: compilation error, runtime error on all tests, runtime error on some tests, wrong answer on all tests, and wrong answer on some tests. This classification offers a novel perspective on evaluating failed examples, as the complexity increases progressively from passing compilation to executing without runtime errors, and finally, producing correct outputs.

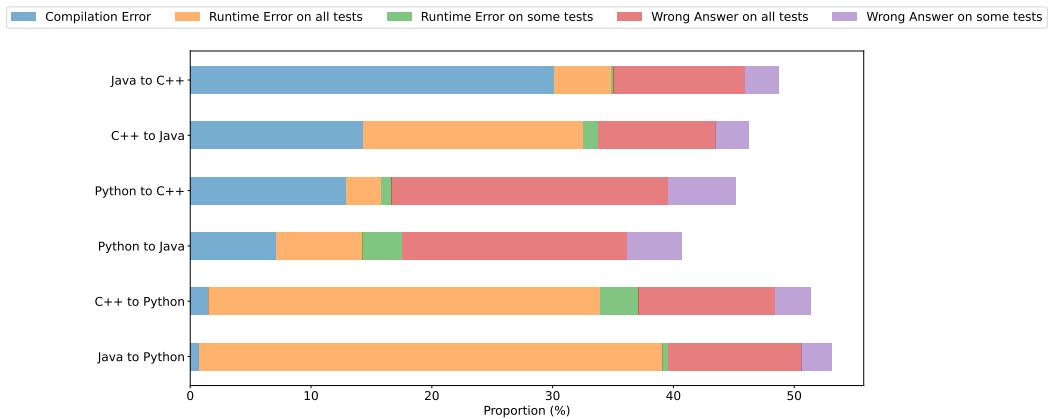


Fig. 2. Distribution of failed samples across five categories in each translation dataset.

The distribution of failed samples across five categories in each translation dataset is illustrated in Figure 2. As shown, the majority of failures vary across different translation datasets. For Python-to-Java and Python-to-C++ translations, wrong answers dominate the failure categories, with a notable proportion of wrong answers on all tests. In contrast, C++-to-Python and Java-to-Python translations exhibit a significantly higher occurrence of runtime errors. This observation aligns with intuition, as Python is a dynamically typed language with more lenient syntax checking, meaning many errors – such as the absence of corresponding functions or type mismatches – only surface during runtime. While examining C++-to-Java and Java-to-C++ translations, compilation errors emerge as the primary source of failure. In addition to the findings mentioned above, we have observed a significant discovery: whether it is a runtime error or a wrong answer, the translated program tends to fail on all tests in most cases. When a translated program fails on all tests, it suggests that the error is “shallow”, meaning that the error is central to the program’s logic or structure rather than specific to certain inputs or edges. Such errors often stem from incorrect syntax, missing functions, or major semantic misinterpretations that prevent any part of the translated program from working as intended. Because “shallow” errors prevent the program from passing any tests, they can be effectively exposed through executing the program on a set of tests.

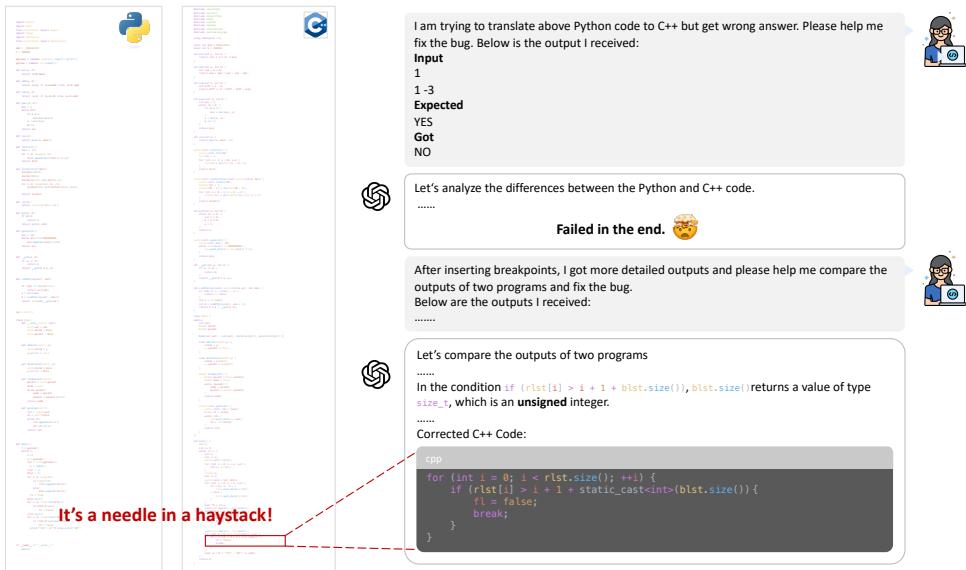


Fig. 3. Code LLMs struggle to repair the translation errors hidden in hundreds of lines of code, but succeed when provided with detailed outputs of breakpoints.

However, identifying and repairing these errors remains a challenge, particularly in the context of translating long code. Recent work [49] suggests that LLMs lack a semantic understanding of how a program executes at run-time, leading to poor performance when instructed to repair the errors when provided with execution results from both the source and translated programs. As illustrated in Figure 3, the Python program is translated to a C++ version but fails to pass the test. Unlike Python, where integers are unbounded, C++ has more strict type limitations. In the conditional statement `if (rlst[i] > i + 1 + blst.size(), blst.size()`, `blst.size()` returns a value of type `size_t`, which is an unsigned integer. When comparing signed and unsigned integers, the C++

compiler converts the signed value to unsigned. This behavior leads to the semantic discrepancy between the Python and C++ programs when encountering negative values in the list.

This example shows that semantic differences between Python and C++ lead to incorrect results. If we could systematically identify these discrepancies and align the program states of the original and translated programs throughout execution, the final outputs would be guaranteed to match, thereby ensuring a correct translation. Inspired by this insight, we apply source code instrumentation – a classical technique in dynamic program analysis – to capture the program states and leverage code LLMs to align the program states from entry to exit. Further details will be provided in the next section.

4 METHOD

The proposed approach PAST consists of three stages: (1) the Instrumentation Stage, which leverages the code LLM to insert breakpoints into the source program in positions that will best reveal the execution flow and variable values, (2) the Translation Stage, which translates the source program into the target language while preserving the breakpoints. If the translated program fails on public tests, it proceeds to (3) the Program State Alignment Stage, where code LLMs are tasked with addressing inconsistencies between program states. In this stage, we employ two steps: direct repair and localize-re-translate. Initially, based on the error messages and the captured program states, we instruct the code LLM to repair the error directly. If this approach is unsuccessful, we localize the erroneous code snippet in the target program and its corresponding snippet in the source program, after that we re-translate the relevant source snippet.

Algorithm 1: Translate with Instrumentation and Program State Alignment

```

Input: source program  $x$ , public_tests  $T$ 
Output: translated program
1  $\hat{x} \leftarrow \text{instrumentation}(x)$                                      ▷ Stage 1: Instrumentation
2  $\hat{y} \leftarrow \text{LLM\_translate}(\hat{x})$                                      ▷ Stage 2: Translation
3 if  $\text{check\_compilation\_error}(\hat{x}) \vee \text{check\_compilation\_error}(\hat{y})$  then
4   | return  $\text{remove\_instrumentation}(\hat{y})$ 
5 end
6  $\text{error}_y, \text{output}_y \leftarrow \text{execute\_on\_tests}(\hat{y}, T)$ 
7  $\text{error}_x, \text{output}_x \leftarrow \text{execute\_on\_tests}(\hat{x}, T)$ 
8 if  $\text{output}_y = \text{output}_x$  then
9   | return  $\text{remove\_instrumentation}(\hat{y})$ 
10 end
11  $\text{diff}_{xy} \leftarrow \text{diff\_by\_line}(\text{output}_y, \text{output}_x)$ 
12  $\hat{y}_{\text{repaired}} \leftarrow \text{LLM\_direct\_repair}(\hat{y}, \text{error}_y, \text{diff}_{xy})$            ▷ Stage 3: Direct Repair
13  $\text{error}_{\text{repaired}}, \text{output}_{\text{repaired}} \leftarrow \text{execute\_on\_tests}(\hat{y}_{\text{repaired}}, T)$ 
14 if  $\text{output}_{\text{repaired}} = \text{output}_x$  then
15   | return  $\text{remove\_instrumentation}(\hat{y}_{\text{repaired}})$ 
16 end
17  $\hat{y}_{\text{localize}} \leftarrow \text{LLM\_localize\_and\_retranslate}(\hat{y}, \text{error}_y, \text{diff}_{xy})$       ▷ Stage 3: Localize and Re-translate
18  $\text{error}_{\text{localize}}, \text{output}_{\text{localize}} \leftarrow \text{execute\_on\_tests}(\hat{y}_{\text{localize}}, T)$ 
19 if  $\text{output}_{\text{localize}} = \text{output}_x$  then
20   | return  $\text{remove\_instrumentation}(\hat{y}_{\text{localize}})$ 
21 end
22 return  $\text{remove\_instrumentation}(\hat{y}_{\text{localize}})$                                      ▷ Failed to Translate

```

As shown in Algorithm 1, our method aims to translate the source program x into the target program y while ensuring that both exhibit consistent behavior on public test cases. The process

begins by instrumenting the source program x to generate the instrumented program \hat{x} . A code LLM is then tasked with translating \hat{x} into \hat{y} in target language. If either \hat{x} or \hat{y} encounters a compilation error, the failed translation results are directly returned, although such cases are relatively rare. Next, \hat{x} and \hat{y} are executed on public test cases to obtain their error message (empty string if no error encountered) and outputs, which are then compared. If the outputs are identical, the instrumentation code in \hat{y} is removed, and the result is returned. If the outputs differ, the line-by-line differences between the outputs are calculated. The LLM is then used to repair \hat{x} based on these differences and the execution result type, producing the repaired program $\hat{y}_{\text{repaired}}$. The program is re-executed on public test cases, and the outputs are compared again. If the outputs are still different, the LLM is employed to localize and repair \hat{y} , producing $\hat{y}_{\text{localize}}$. If the outputs of $\hat{y}_{\text{localize}}$ and \hat{x} remain inconsistent, the method returns the failed translation result.

4.1 The Instrumentation Stage

Instrumentation is widely employed to analyze program runtime states, and we leverage it here to capture and describe these states effectively. The process begins by inserting breakpoints at key points in the code to track the program's state. To accomplish this, we utilize the code LLM to identify the crucial positions and insert print statements to output variable values, and an example is illustrated in Figure 4. To prevent an overload of overwhelming breakpoints and outputs, we emphasize that the information that breakpoints display should be relevant to understanding the program's logic and flow. Furthermore, to ensure that the breakpoint outputs can effectively help in identifying errors, we instruct the code LLM to format the breakpoint outputs as: "[description of the current position]: [variable_name] = [variable_value]". This format ensures the breakpoint outputs are clear and structured, facilitating more precise localization of program state inconsistencies in the Program State Alignment Stage.

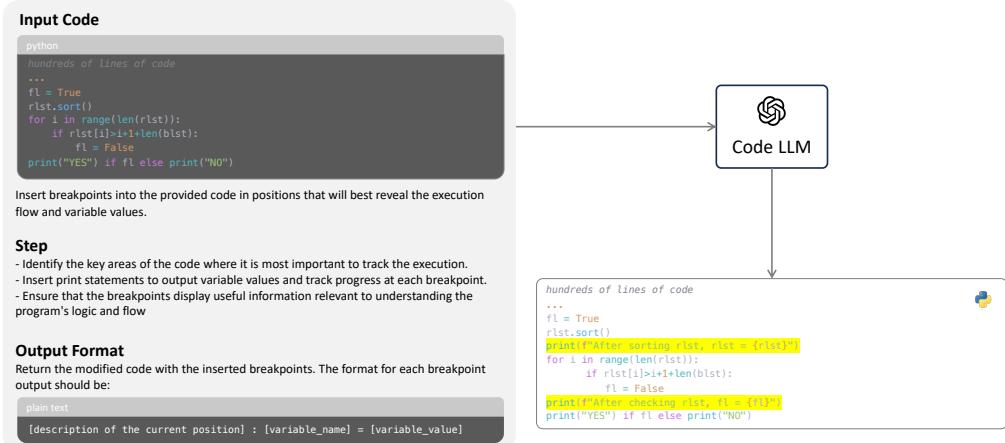


Fig. 4. An Example of the Instrumentation Stage.

4.2 The Translation Stage

After the Instrumentation Stage, the source program is now augmented with strategically placed breakpoints that facilitate monitoring key variables and execution traces. In the Translation Stage,

the primary task is to translate the source program into the target language while preserving the inserted breakpoints. The prompt can be formally defined using Ninja [61], as shown in Figure 5, where lang1 refers to the source language, lang2 refers to the target language, and code represents the source program with breakpoints. With this prompt template, the code LLM can translate the source program while ensuring the semantics of the breakpoint statements remain unchanged.

```
# Input Code
```{{ lang1 }}
{{ code }}
```

Translate the provided {{ lang1 }} code to {{ lang2 }} while preserving the functionality and logic of the original code. Keep all breakpoint statements that print values to stdout intact, ensuring they function in the translated version as they do in the original.

# Step
- Analyze the provided code to fully understand its structure, logic, and functionality.
- Identify any syntax and language-specific constructs that need to be adapted for {{ lang2 }}.
- Translate the code, ensuring all comments, variable names, and logic remain consistent with the original.
- Ensure that any breakpoint statements that print values to stdout are accurately translated to {{ lang2 }}.

# Output Format
Provide the translated code without any additional commentary.
```

Fig. 5. Prompt template for translating source program with breakpoints.

4.3 The Program State Alignment Stage

After the Translation Stage, we obtain the source and target programs both inserted with breakpoints. We then utilize execution to provide a comprehensive view of any semantic discrepancies between the source and target programs. This process involves executing both programs with identical test inputs and comparing their intermediate states and final outputs, as illustrated in the upper left corner of Figure 6. The translation is considered successful if the target program produces intermediate states and outputs identical to those of the source program on public tests. If discrepancies are detected, we proceed to the next step, where two repair steps are employed to address the inconsistencies between the original and translated programs.

The first repair step is direct repair. During execution, the program generates an error message alongside its output, which may indicate a compilation error, runtime error, or wrong answer. The program output includes both breakpoint outputs, formatted as described in Section 4.1, and the final output. The breakpoint outputs include descriptions of the positions and variable values, allowing us to identify inconsistencies in the program states as soon as the breakpoint outputs of both programs diverge. We employ text diff tools to compare the program outputs, highlighting the difference and minimizing redundant, irrelevant content to better focus the code LLM’s attention. The source program, target program, error message, and output differences are provided to the code LLM, which is tasked with aligning program states based on this information. While direct repair appears intuitive, it has proven to be highly effective when supplemented with breakpoint outputs. We will demonstrate the effectiveness of direct repair in our experiments.

The second repair step is localize-and-re-translate. This step is motivated by the observation that code LLMs tend to struggle with translating entirely programs accurately but excel when focused on shorter code snippets. Therefore, we first direct the code LLM to localize the translation error and extract the erroneous code snippet in the target language, alongside its corresponding snippet in the source language – referred to as the target code snippet and source code snippet, respectively. We then re-translate the source code snippet and reintegrate the new target snippet back into the

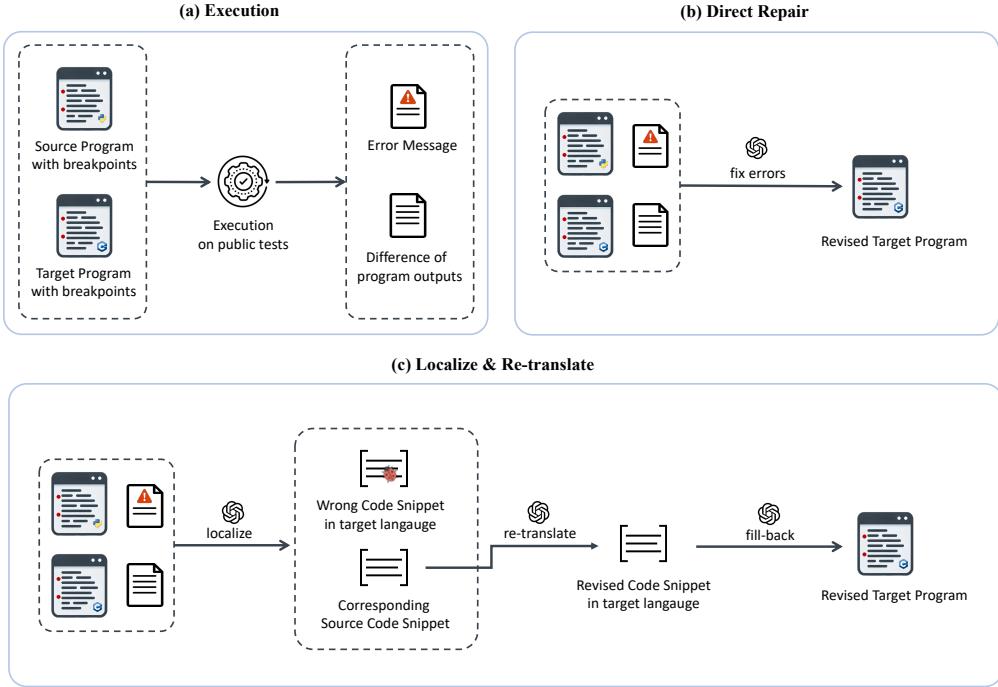


Fig. 6. Illustration of the Program State Alignment Stage.

target program. By eliminating the need to distribute attention across lengthy input, the code LLM can better address subtle semantic inconsistencies. Consequently, localize-and-re-translate can further enhance translation accuracy beyond what is achieved through direct repair.

Finally, the translated program will be executed on private tests to verify that it produces outputs identical to those of the original program. The private tests included in the benchmark are designed to be as exhaustive as possible, ensuring that translated programs cannot be incorrectly marked as correct due to insufficient test coverage. This rigorous testing approach helps maintain the integrity of the evaluation process.

5 BENCHMARK

In this section, we provide detailed descriptions of our proposed *LongTrans* benchmark, how it was collected and processed, and its comparison with other code translation benchmarks. The *LongTrans* benchmark consists of programs from three popular programming languages: C++, Java, and Python, and utilizes sufficient tests to evaluate translation results based on execution results. The detailed statistics about the *LongTrans* dataset are presented in Table 1.

5.1 Data Collection and Processing

Our dataset was collected from the CodeContests dataset [37], which consists of programming problems collected from Codeforces, LeetCode, and other online judge platforms. In addition to the problems themselves, the dataset includes corresponding solutions and high-quality test cases.

Table 1. Statistics for the *LongTrans* benchmark. The number of tokens is based on the RoBERTa [40] tokenizer. Length is the number of characters.

| Split | Language | Samples | Avg. Tokens per Sample | Avg. Length | Avg. Tests per Sample |
|-------|----------|---------|------------------------|-------------|-----------------------|
| Train | C++ | 96791 | 775 | 1465 | 99 |
| | Java | 62994 | 1919 | 3578 | 112 |
| | Python | 58517 | 429 | 785 | 116 |
| Valid | C++ | 12734 | 793 | 1519 | 202 |
| | Java | 8251 | 2155 | 3919 | 198 |
| | Python | 8396 | 711 | 1294 | 199 |
| Test | C++ | 14677 | 825 | 1585 | 201 |
| | Java | 8894 | 2379 | 4342 | 197 |
| | Python | 8248 | 619 | 1147 | 197 |

This enables us to evaluate translated programs through execution instead of relying on lexical similarity, which has been proven to be a poor proxy for functional equivalence [9].

By collecting submitted solutions from the CodeContests dataset, we obtained adequate samples written in the most popular submission languages: C++, Python, and Java. However, we identified a significant number of false positive samples, where incorrect submissions are erroneously marked as correct due to insufficient test coverage. To mitigate this issue, we exclusively select samples collected from Codeforces, where the average number of tests per problem is notably higher than other sources. Moreover, we observed that some programming problems had many identical or nearly identical submissions, leading to an abundance of near-duplicate samples. To address this, we employed the MinHash [7] algorithm for deduplication, effectively reducing redundancy. Furthermore, we performed clustering within each problem’s submissions to identify and select representative samples. This approach minimized the impact of near-duplicate submissions on the overall quality and diversity of the benchmark.

Table 2. Comparison of *LongTrans* with existing code translation datasets. We report Samples and Avg. Tokens per Sample for Train/Valid/Test sets of each dataset. For Samples, we report **program-level** counts for AVATAR, CoST, and XLCoST.

| Dataset | Unit Tests | Languages | Samples | Avg. Tokens per Sample | Avg.Length |
|------------------------|------------|-------------------|------------------------|------------------------|--------------------|
| CodeXGLUE [43] | ✗ | Java, C# | 10300 / 500 / 1000 | 59 / 63 / 58 | 205 / 218 / 202 |
| AVATAR [3] | ✓ | Java, Python | 7113 / 476 / 1906 | 239 / 235 / 234 | 691 / 687 / 688 |
| TransCoder [54] | ✓ | C++, Java, Python | - / 470 / 948 | - / 119 / 120 | - / 313 / 311 |
| CoST [77] | ✗ | 7 languages | 14260 / 1029 / 1449 | 272 / 180 / 199 | 770 / 458 / 511 |
| XLCoST [76] | ✗ | 7 languages | 102559 / 5892 / 10632 | 234 / 232 / 222 | 644 / 634 / 606 |
| MultilingualTrans [69] | ✗ | 9 languages | 19115 / 3759 / 7545 | 398 / 421 / 491 | 1099 / 1135 / 1358 |
| NicheTrans [69] | ✗ | 37 languages | 165457 / 23509 / 47502 | 292 / 375 / 505 | 785 / 995 / 1372 |
| LLMTrans [69] | ✓ | 8 languages | - / - / 350 | - / - / 270 | - / - / 745 |
| CoTran [28] | ✓ | Java, Python | 55178 / 442 / 1745 | 288 / 242 / 249 | 619 / 639 / 654 |
| <i>LongTrans</i> | ✓ | C++, Java, Python | 218302 / 29381 / 31819 | 1012 / 1152 / 1206 | 1893 / 2129 / 2243 |

5.2 Benchmark Comparisons

As indicated by its name, *LongTrans* is distinguished by the average length of its samples. As shown in table 2, the average number of tokens per sample in *LongTrans* is approximately 2-3 times longer than in previous benchmarks, presenting a significant challenge for code LLMs. The unit tests mean

whether or not to run the generated code when testing, for example, computing the BLEU metric does not require execution, but computing the accuracy does.

6 EXPERIMENT SETTING

In this section, we will describe our experimental setup, which consists of the studied large language models, evaluation metrics, implementation of the experiment, and research questions.

6.1 Studied Models

We introduce different families of recent LLMs of diverse sizes for empirical study. Their detailed information is listed below.

- **CodeGeeX4** [75]: is the open-source version of the latest CodeGeeX4 model series. It is a multilingual code generation model continually trained on the GLM-4 [21], significantly enhancing its code generation capabilities. It is developed upon the ChatGLM2 [16] architecture and is enhanced with an extensive dataset of coding examples.
- **CodeGemma** [60]: is a collection of specialized open code models built on top of Gemma, capable of a variety of code and natural language generation tasks. CodeGemma 7B pre-trained (PT) and instruction-tuned (IT) variants have remarkably resilient natural language understanding, excel in mathematical reasoning, and match the code capabilities of other open-source models.
- **CodeQwen1.5**³: is the Code-Specific version of Qwen1.5 [6]. It is a transformer-based decoder-only language model pre-trained on a large amount of data of codes. CodeQwen1.5 is trained on 3 trillion tokens of data of codes, and it includes group query attention [4] to reduce KV cache and accelerate inference.
- **Qwen2.5** [70]: is the latest series of Qwen large language models. Qwen2.5 is pre-trained on their large-scale dataset, encompassing up to 18 trillion tokens. It has acquired significantly more knowledge and has greatly improved capabilities in coding and mathematics compared to its preceding series.
- **StarCoder2** [41]: is a publicly accessible model with a substantial parameter count of 15 billion. It is pre-trained on over 600 programming languages from The Stack v2, as well as natural language text from sources like Wikipedia, Arxiv, and GitHub issues, ensuring proficiency across a wide range of coding tasks.
- **Codestral**⁴: is trained on a diverse dataset of 80+ programming languages, including the most popular ones, such as Python, Java, C, C++, JavaScript, and Bash. It also performs well on more specific ones like Swift and Fortran. This broad language base ensures that Codestral can assist developers in various coding environments and projects.
- **Mistral-Large-2**⁵: is the new generation of flagship model developed by Mistral, boasting 123 billion parameters. Compared to its predecessor, Mistral Large 2 is significantly more capable in code generation, mathematics, and reasoning.
- **DeepSeek-Coder** [23]: is a range of open-source code models with sizes from 1.3B to 33B. It is pre-trained from scratch on 2 trillion tokens, with a composition of 87% code and 13% natural language in both English and Chinese. For coding capabilities, DeepSeek-Coder achieves state-of-the-art performance among open-source code models on multiple programming languages and various benchmarks.

³<https://huggingface.co/Qwen/CodeQwen1.5-7B>

⁴<https://mistral.ai/news/codestral/>

⁵<https://mistral.ai/news/mistral-large-2407/>

- **DeepSeek-Coder-V2 [78]**: is an open-source Mixture-of-Experts (MoE) code language model that achieves performance comparable to GPT-4 in code-specific tasks. Built on the DeepSeekMoE [12] framework, DeepSeek-Coder-V2 comprises 236 billion parameters, with 21 billion activated during inference.
- **LlaMA-3.1 [62]**: A family of multilingual large language models ranging from 8B to 405B. We include two versions of 8B/70B of LlaMA-3.1 for experiments. The Llama 3.1 instruction-tuned, text-only models are optimized for multilingual dialogue and excel on common industry benchmarks, outperforming many open-source and closed chat models.
- **CodeLlaMA [53]**: encompasses a series of code-centric Large Language Models (LLMs) that are derivatives of LlaMA2 [63]. Available in three sizes – 7B, 13B, and 34B – these models undergo continued training on a vast 500 billion token code corpus, building upon the foundational LlaMA2 architecture.
- **GPT-4o and GPT-4o-mini [1]** are advanced generative AI models developed by OpenAI. While they are not explicitly trained for code generation, they also demonstrate notable performance in this domain. Their effectiveness in handling code generation tasks is largely attributed to their massive scale in terms of parameter count.

6.2 Evaluation Metrics

Following the previous studies in the code translation field [54, 56, 58, 72], we adopt Computational Accuracy (CA) to assess the performance of each model.

The computational accuracy computes the ratio that the translated codes can produce the same execution results as the ground truth source code while giving the same inputs [72]. This metric measures the semantic consistency of two programs, assuming that the given example covers all boundary conditions, which can be formalized as:

$$CA = \frac{\sum_{k=1}^N ca(y_k, \hat{y}_k)}{N}, \text{ where} \quad (1)$$

$$ca(y_k, \hat{y}_k) = \begin{cases} 1 & Exec_k(y_k) = Exec_k(\hat{y}_k) \\ 0 & Exec_k(y_k) \neq Exec_k(\hat{y}_k) \end{cases} \quad (2)$$

N denotes the total number of translation samples, y_k denotes the ground truth of the k -th sample, and \hat{y}_k denotes the translated program via a certain transpiler for the k -th sample. $Exec_k(\cdot)$ denotes the execution result of a program with the test suite of the k -th sample. For example, even if y_k and \hat{y}_k are not identical literally, they are considered a correct translation in CA, as long as $Exec_k(y_k) = Exec_k(\hat{y}_k)$.

6.3 Implementation

We access GPT-4o and GPT-4o-mini via OpenAI’s API⁶, while for open-source code LLMs, we load their weights from Huggingface⁷ and perform inference using vllm [34], which effectively accelerates the inference with paged attention. To ensure fairness, we maintain consistent default settings for all code LLMs in the empirical study. We use nucleus sampling [25] with parameters set to $top_p = 0.95$, $temperature = 0.2$. We adopt a zero-shot learning setting, as the code LLMs we investigated are instruction fine-tuned [51], and this setting closely aligns with real-world scenarios. For our approach, the sampling settings of backbone LLMs in three stages are kept as default.

⁶<https://platform.openai.com/>

⁷<https://huggingface.co/>

6.4 Research Questions and Evaluation Methodology

In the next section, we will introduce five research questions of this work and show the evaluation results.

RQ1: What is the performance of recent LLMs at different scales in long code translation in terms of computation accuracy? This RQ extensively assesses the code translation performance of recent LLMs. In this empirical study, we evaluate the models mentioned in Section 6.1 on six translation datasets of *LongTrans*, i.e., C++ to Python, Python to C++, Java to C++, C++ to Java, Java to Python, and Python to Java. In those datasets, we quantitatively explore the practicality of LLMs in the long code translation task.

RQ2: How does our approach perform with different LLMs? This RQ aims to examine the effectiveness of our approach and its generalization capability on various LLMs. In experiments, the DeepSeekCoder-V2, Qwen2.5, Mistral-Large-2, GPT-4o-mini, and the state-of-the-art LLM, GPT-4o are selected for investigation.

RQ3: How effective are the different components of the method? Beyond translation, our approach consists of three components to boost the code translation, including instrumentation stage, direct-repair step and localize & re-translation step. The ablation study of the different components is designed to investigate their contributions in terms of computation accuracy on five LLMs including DeepSeekCoder-V2, Qwen2.5, Mistral-Large-2, GPT-4o-mini, and GPT-4o.

RQ4: How about LLM translation performance as code length increases?

In this RQ, we investigate the impact of code length on the translation performance of code LLMs, specifically analyzing how variations in the length of input code affect the computational accuracy of the generated outputs. We conduct experiments on six translation datasets from the *LongTrans* benchmark, comparing the performance of code LLMs and their enhanced version with our approach, and present the average results. These datasets are categorized by source code length into five distinct ranges for evaluation: less than 256 tokens, 256-512 tokens, 512-1024 tokens, 1024-2048 tokens, and more than 2048 tokens.

RQ5: How about the sensitivity of the approach to the quality of the public tests? The quality of public tests impacts the efficiency of code LLMs, as tests with low coverage may not reveal potential issues. To further demonstrate the generality of our approach, we examine its sensitivity to the quality of the public tests. By varying the quality of public tests to adjust the executed feedback quality, we aim to explore whether our approach remains effective even when the executed feedback quality is of lower quality.

7 EXPERIMENT RESULTS

7.1 RQ1: What is the performance of recent LLMs at different scales in long code translation in terms of computation accuracy?

Table 3 presents the empirical results of each model on the *LongTrans* benchmark, with the performance measured in terms of Correctness Accuracy. (1) As the results clearly indicate, GPT-4o consistently performs the best on most translation datasets, achieving accuracy rates of 59.18%, 50.10%, 61.70%, and 56.93% in the C++ to Java, C++ to Python, Java to Python, and Python to C++ datasets. Additionally, it is surprising that DeepSeek-Coder outperforms GPT-4o in the Java to C++ and Python to Java datasets. (2) Overall, as the model parameter size increases, the translation performance of large language models gradually improves. (3) In long code translation scenarios, large language models exhibit better average performance when translating from Java to C++. This may be due to the structural similarities between Java and C++, making the mappings more explicit in long code translations. In contrast, the average performance for translations from C++ to Python is relatively lower, likely because of the significant syntactic and semantic differences between C++

Table 3. The empirical results of LLMs across different programming languages.

| Model | Language | | | | | | Average |
|----------------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| | C++-Java | C++-Python | Java-C++ | Java-Python | Python-C++ | Python-Java | |
| CodeGeeX4-9B | 35.77% | 28.09% | 35.37% | 40.09% | 37.52% | 38.78% | 35.94% |
| CodeGemma-7B | 14.90% | 19.46% | 31.70% | 20.67% | 26.58% | 27.00% | 23.39% |
| CodeQwen1.5-7B | 24.36% | 21.13% | 31.25% | 23.71% | 31.64% | 32.45% | 27.42% |
| StarCoder2-15B | 30.39% | 26.67% | 46.34% | 36.72% | 37.63% | 34.41% | 35.36% |
| Codestrail-22B | 43.59% | 38.16% | 47.70% | 41.80% | 51.21% | 53.07% | 45.92% |
| Mistral-Large-2-123B | 48.31% | 46.18% | 54.47% | 58.11% | 50.73% | 46.25% | 50.68% |
| CodeLlama-7B | 18.34% | 12.46% | 24.25% | 15.95% | 13.65% | 13.40% | 16.34% |
| CodeLlama-13B | 17.51% | 18.21% | 26.50% | 19.42% | 15.65% | 15.17% | 18.74% |
| CodeLlama-34B | 21.19% | 11.79% | 21.87% | 6.91% | 17.70% | 15.48% | 15.82% |
| DeepSeekCoder-1.3B | 21.47% | 15.45% | 30.02% | 18.19% | 24.19% | 15.04% | 20.73% |
| DeepSeekCoder-6.7B | 40.03% | 27.47% | 42.91% | 34.13% | 46.19% | 45.26% | 39.33% |
| DeepSeekCoder-33B | 49.61% | 40.10% | 63.90% | 52.77% | 54.46% | 52.71% | 52.26% |
| DeepSeekCoder-V2 | 53.78% | 48.64% | 51.28% | 46.89% | 54.81% | 59.31% | 52.45% |
| Llama3.1-8B | 32.62% | 27.36% | 29.08% | 30.33% | 35.26% | 37.21% | 31.98% |
| Llama3.1-70B | 50.18% | 43.70% | 49.71% | 47.72% | 53.30% | 55.42% | 50.01% |
| GPT-4o-mini | 45.66% | 45.11% | 62.29% | 61.70% | 52.06% | 46.71% | 52.26% |
| GPT-4o | 59.18% | 50.10% | 62.17% | 60.99% | 56.93% | 55.71% | 57.51% |
| Average | 35.70% | 30.59% | 41.81% | 36.24% | 38.79% | 37.85% | 36.83% |

For clarification, we adopt “-” to concatenate LLM and each of its corresponding parameter sizes for discrimination. For example, Llama3.1 with 70B parameters is dubbed Llama3.1-70B. An exception is that DeepSeek-Coder-V2 has 236B parameters in total, with 21B activated during inference. Bold indicates the optimal value on the current dataset.

and Python. These differences lead to unclear structural correspondences, particularly in long code translations, which can result in mistranslations and omissions.

Answer to RQ1: The experimental results demonstrate that larger model sizes correlate with improved translation accuracy, especially in long code translation. Models also translate Java to C++ more effectively, likely due to structural similarities, whereas C++ to Python translations suffer from syntactic and semantic differences, leading to more frequent errors and omissions.

7.2 RQ2: How does our approach perform with different LLMs?

In order to verify the general applicability of our method, we conducted validation on multiple LLMs with different architectures and different parameters, and the results are shown in Table 4. As can be seen, our approach consistently boosts the performance of LLMs, including DeepSeekCoder-V2, Qwen2.5, Mistral-Large-2, GPT-4o-mini, and GPT-4o, of 46.64%, 49.65%, 38.80%, 45.32%, and 47.95% improvement in terms of computation accuracy on average. This demonstrates the effectiveness of our approach for improving long code translation of LLMs with different architectures and different parameters. Particularly, Table 4 demonstrates the translation on GPT-4o achieves the computation accuracy of 84.70% on average, which is the highest among all experimented LLMs.

Answer to RQ2: Our method consistently improves the long code translation performance across various large language models with different architectures and parameters. This confirms the effectiveness of our approach for enhancing long code translation in diverse LLMs.

Table 4. Comparison of the performance of LLMs of code with and without PAST.

| Model | C++ | | Java | | Python | | Average |
|--------------------------------|--------------------|---------|---------|---------|---------|---------|---------|
| | Java | Python | C++ | Python | C++ | Java | |
| DeepSeekCoder-V2
+ PAST | 53.78% | 48.64% | 51.28% | 46.89% | 54.81% | 59.31% | 52.45% |
| | 78.10% | 74.27% | 74.12% | 80.92% | 71.83% | 79.33% | 76.43% |
| | <i>Improvement</i> | +45.22% | +52.69% | +44.54% | +72.57% | +31.05% | +33.75% |
| Qwen-2.5-72B
+ PAST | 38.52% | 39.25% | 42.33% | 51.40% | 50.37% | 53.69% | 45.93% |
| | 66.34% | 63.52% | 55.17% | 72.83% | 69.26% | 82.84% | 68.49% |
| | <i>Improvement</i> | +72.22% | +61.83% | +30.33% | +41.69% | +37.50% | +54.29% |
| Mistral-Large-2-123B
+ PAST | 48.31% | 46.18% | 54.47% | 58.11% | 50.73% | 46.25% | 50.68% |
| | 72.13% | 65.80% | 65.05% | 68.42% | 70.78% | 76.00% | 69.67% |
| | <i>Improvement</i> | +49.31% | +42.49% | +19.42% | +17.74% | +39.52% | +64.32% |
| GPT-4o-mini
+ PAST | 45.66% | 45.11% | 62.29% | 61.70% | 52.06% | 46.71% | 52.26% |
| | 70.02% | 59.26% | 78.76% | 72.69% | 82.78% | 85.93% | 74.91% |
| | <i>Improvement</i> | +53.35% | +31.37% | +26.44% | +17.81% | +59.01% | +83.96% |
| GPT-4o
+ PAST | 59.18% | 50.10% | 62.17% | 60.99% | 56.93% | 55.71% | 57.51% |
| | 86.92% | 78.13% | 81.79% | 82.92% | 86.42% | 92.24% | 84.70% |
| | <i>Improvement</i> | +46.87% | +55.95% | +31.56% | +35.96% | +51.80% | +65.57% |

7.3 RQ3: How effective are the different components of the method?

As shown in Table 5, the ablation study is conducted on three important components, instrumentation stage, direct-repair step, and localize & re-translation step of the PAST.

By examining the experimental results, we observe that these three components enhance the translation performance of LLMs of code across different parameter scales and translation datasets, highlighting their generalizability. First, The most significant improvement emerges when direct repair is integrated with instrumentation, yielding an absolute improvement in computational accuracy of 12% to 21% compared to direct repair without instrumentation across translation datasets. This suggests that for translating lengthy code, instrumentation greatly enhances LLMs of code in capturing and aligning the program states during execution, thus achieving better performance, which aligns with our research motivation. Additionally, localizing inconsistencies in program states and precisely guiding the LLMs of code to re-translate further enhances translation performance, with an improvement of 1%-4% in computational accuracy across translation datasets. Although this improvement may seem modest in absolute terms, it is achieved on top of already high performance brought by direct repair and instrumentation. This makes the enhancement particularly noteworthy and underscores the effectiveness of the localize-and-re-translate step.

Answer to RQ3: The experimental results reveal that these three components significantly enhance translation performance across code LLMs of varying scales and multiple language pairs, demonstrating broad generalizability. Especially, the results show that integrating instrumentation dramatically improves translation performance, highlighting that capturing and aligning program states plays a vital role in achieving better code translation.

7.4 RQ4: How about LLM translation performance as code length increases?

To further validate the effectiveness of PAST, we evaluated code LLM performance on different input lengths. The results show the average computation accuracy of code translation across six datasets. These datasets are grouped by source code length into five ranges for evaluation: less than 256 tokens, 256-512 tokens, 512-1024 tokens, 1024-2048 tokens, and more than 2048 tokens.

Table 5. Model performance across different programming languages and improvements with techniques such as direct repair (R), instrumentation (I), and localization & re-translation ($L\&R$).

| Model | C++ | | Java | | Python | | Average |
|----------------------|--------|--------|--------|--------|--------|--------|---------|
| | Java | Python | C++ | Python | C++ | Java | |
| DeepSeekCoder-V2 | 53.78% | 48.64% | 51.28% | 46.89% | 54.81% | 59.31% | 52.45% |
| | 59.57% | 54.16% | 53.64% | 49.65% | 58.06% | 60.43% | 56.48% |
| | 75.62% | 72.09% | 73.02% | 77.56% | 70.40% | 77.49% | 74.36% |
| | 78.10% | 74.27% | 74.12% | 80.92% | 71.83% | 79.33% | 76.43% |
| Qwen2.5-72B | 38.52% | 39.25% | 42.33% | 51.40% | 50.37% | 53.69% | 45.93% |
| | 47.80% | 45.50% | 43.14% | 56.67% | 54.03% | 57.84% | 50.83% |
| | 63.33% | 60.80% | 51.05% | 68.55% | 66.84% | 80.08% | 65.11% |
| | 66.34% | 63.52% | 55.17% | 72.83% | 69.26% | 82.84% | 68.33% |
| Mistral-Large-2-123B | 48.31% | 46.18% | 54.47% | 58.11% | 50.73% | 46.25% | 50.68% |
| | 57.35% | 50.79% | 55.92% | 59.57% | 56.30% | 48.25% | 54.70% |
| | 70.06% | 63.84% | 61.02% | 66.29% | 68.85% | 73.57% | 67.27% |
| | 72.13% | 65.80% | 65.05% | 68.42% | 70.78% | 76.00% | 69.70% |
| GPT-4o-mini | 45.66% | 45.11% | 62.29% | 61.70% | 52.06% | 46.71% | 52.26% |
| | 53.94% | 46.84% | 66.09% | 63.06% | 60.89% | 49.79% | 56.77% |
| | 67.30% | 56.88% | 77.64% | 70.56% | 78.90% | 84.24% | 72.59% |
| | 70.02% | 59.26% | 78.76% | 72.69% | 82.78% | 85.93% | 74.91% |
| GPT-4o | 59.18% | 50.10% | 62.17% | 60.99% | 56.93% | 55.71% | 57.51% |
| | 66.16% | 58.09% | 65.52% | 62.91% | 66.73% | 58.19% | 62.93% |
| | 85.63% | 77.11% | 81.01% | 81.80% | 84.60% | 91.50% | 83.61% |
| | 86.92% | 78.13% | 81.79% | 82.92% | 86.42% | 92.24% | 84.74% |

Table 6 demonstrates that translation performance for code LLMs of different scales declines as code length increases, likely due to the greater complexity of translating longer code sequences. PAST, however, can partially offset this performance decline, resulting in a more gradual decrease in performance with increasing code length. Moreover, the effectiveness of PAST in mitigating performance loss becomes increasingly pronounced with longer code, as indicated by progressively higher relative improvement rates. Notably, for code lengths exceeding 2048 tokens, PAST achieves a remarkable relative improvement of 97.8% in computation accuracy for DeepSeekCoder-V2 and 70.20% for GPT-4o. This highlights PAST’s effectiveness and generalizability in addressing the challenges associated with long code translation.

Answer to RQ4: The experimental results show that PAST significantly improves average computation accuracy across six datasets. While translation performance for code LLMs declines as code length increases, PAST effectively mitigates this drop, demonstrating its effectiveness and generalizability in handling long-code translation challenges.

7.5 RQ5: How about the sensitivity of the approach to the quality of the public tests?

The quality of public tests can significantly impact the performance of our approach. Public tests with low coverage may fail to reveal potential issues in the Program State Alignment Stage, leading to erroneous programs passing these tests but subsequently failing in the final evaluation based on private tests. To evaluate the sensitivity of our approach to the quality of the public tests, we utilize code LLMs to generate tests to replace the public tests provided in the benchmark. Specifically, we instruct code LLMs to generate five tests per sample based on the original program. In this context, the code LLMs are primarily tasked with designing the test inputs given the original program.

Table 6. Average computation accuracy of different LLMs across different input sizes.

| Model | < 256 | 256 - 512 | 512 - 1024 | 1024 - 2048 | > 2048 |
|--------------------------------|---------|-----------|------------|-------------|---------|
| DeepSeek-Coder-V2
+ PAST | 69.11% | 60.91% | 46.24% | 33.45% | 26.83% |
| | 89.46% | 84.58% | 75.13% | 64.58% | 53.08% |
| | +29.45% | +38.86% | +62.48% | +93.06% | +97.84% |
| Qwen2.5-72B
+ PAST | 63.87% | 50.85% | 33.28% | 22.58% | 18.99% |
| | 84.62% | 72.73% | 57.73% | 44.20% | 33.20% |
| | +32.49% | +43.03% | +73.47% | +95.75% | +74.83% |
| Mistral-Large-2-123B
+ PAST | 63.98% | 60.33% | 46.24% | 34.00% | 26.84% |
| | 81.39% | 78.57% | 71.36% | 59.51% | 39.44% |
| | +27.21% | +30.23% | +54.33% | +75.03% | +46.94% |
| GPT-4o-mini
+ PAST | 62.25% | 54.69% | 44.24% | 40.58% | 33.85% |
| | 86.99% | 77.33% | 66.00% | 58.51% | 42.15% |
| | +39.74% | +41.40% | +49.19% | +44.18% | +24.52% |
| GPT-4o
+ PAST | 67.75% | 59.93% | 52.79% | 46.26% | 34.23% |
| | 92.43% | 88.94% | 81.56% | 76.55% | 58.26% |
| | +36.43% | +48.41% | +54.50% | +65.48% | +70.20% |

Comparing the performance of PAST under public tests of varying quality reveals that the tests generated by code LLMs are of insufficient quality. However, as shown in Table 7, the decline in performance is relatively minor compared to the substantial improvements achieved through our PAST approach. This indicates that while the quality of public tests is important, the robustness of our approach remains largely effective even in less-than-ideal testing conditions.

Table 7. The sensitivity of our approach to the quality of the public tests.

| Model | Public Tests | C++ | | Java | | Python | | Average |
|--|----------------|--------|--------|--------|--------|--------|--------|---------|
| | | Java | Python | C++ | Python | C++ | Java | |
| DeepSeekCoder-V2
+PAST
+PAST | / | 53.78% | 48.64% | 51.28% | 46.89% | 54.81% | 59.31% | 52.45% |
| | LLM-generated | 74.25% | 72.28% | 69.10% | 75.17% | 65.09% | 77.94% | 72.31% |
| | Human-designed | 78.10% | 74.27% | 74.12% | 80.92% | 71.83% | 79.33% | 76.43% |
| Qwen2.5-72B
+PAST
+PAST | / | 38.52% | 39.25% | 42.33% | 51.40% | 50.37% | 53.69% | 45.93% |
| | LLM-generated | 63.48% | 61.64% | 51.01% | 68.53% | 66.30% | 81.57% | 65.42% |
| | Human-designed | 66.34% | 63.52% | 55.17% | 72.83% | 69.26% | 82.84% | 68.33% |
| Mistral-Large-2-123B
+PAST
+PAST | / | 48.31% | 46.18% | 54.47% | 58.11% | 50.73% | 46.25% | 50.68% |
| | LLM-generated | 68.25% | 62.67% | 61.34% | 67.97% | 69.33% | 76.36% | 67.65% |
| | Human-designed | 72.13% | 65.80% | 65.05% | 68.42% | 70.78% | 76.00% | 69.70% |

?

Answer to RQ5: The quality of public tests affects translation performance; LLM-generated tests provided weaker feedback in the Program State Alignment Stage, resulting in lower translation accuracy. Nevertheless, PAST still demonstrates a significant improvement in code LLMs across all datasets. This demonstrates the robustness and efficiency of our approach under public tests of varying quality.

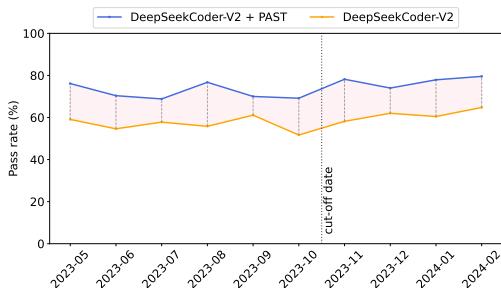
8 DISCUSSION

8.1 Impact of Data Contamination

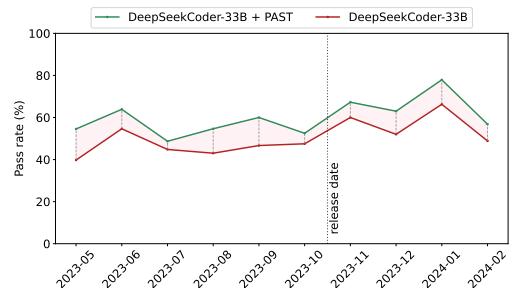
Data contamination arises when instances from a test set are inadvertently included in the training set, compromising the separation between the two datasets. Given that the pre-training and fine-tuning of code LLMs requires a vast amount of code corpus, it is plausible that some translation instances in the test split of our benchmark may have been used during the training of the code LLMs. This overlap could result in an overestimation of the performance of code LLMs.

To address this concern, we conduct an experiment to evaluate the impact of data contamination on the effectiveness of our proposed method. We collect 190 programming problems and 515 solutions from the AtCoder⁸ platform, along with their corresponding test cases from LiveCodeBench [?]. Specifically, we gather data from 39 contests held on AtCoder between May 2023 and February 2024. Among the collected solutions, 146 are implemented in C++, 183 in Java, and 186 in Python.

For model selection, since the knowledge cut-off dates for Qwen2.5-72B and Mistral-Large-2-123B are not specified in their respective papers, we selected DeepSeekCoder-V2 for our experiment. The knowledge cut-off date for DeepSeekCoder-V2 is November 2023. As shown in Figure 7a, the performance benefit of our approach remains consistent both before and after the knowledge cut-off date. To further mitigate the potential impact of data contamination introduced during instruction tuning and other processes after pre-training, which could result in a later knowledge cut-off date, we conducted an additional experiment using DeepSeekCoder-33B, whose model weights were released in November 2023. As demonstrated in Figure 7b, the DeepSeekCoder-33B performs consistently better combined with our method. From these two observations, we can confidently conclude that data contamination does not affect the effectiveness of our method.



(a) DeepSeekCoder-V2 w. PAST performs better both before and after the knowledge cut-off date.



(b) The performance benefit also remains consistent before and after the release date of model weights.

8.2 Case Study

To demonstrate our method’s ability to solve real-world translation problems, we translate a real-world open-source project, the Apache Commons Validator⁹, from Java to Python. For repository-level code translation tasks, directly translating the entire repository can be a long shot and lead to numerous failures. To avoid this, we follow [?] to utilize static analysis to split the repository into code fragments and translate fragments separately. Our approach successfully translates 52 fragments that pass all tests with DeepSeekCoder-V2, outperforming the baseline, which translates only 37 fragments correctly. As shown in Figure 8, our approach enables the LLM to identify subtle

⁸<https://atcoder.jp/>

⁹<https://github.com/apache/commons-validator>

differences between APIs through implementation. While this is still a small step toward fully automated repository translation, it marks significant progress toward that goal.

The original Java code:

```

public final class IBANCheckDigit implements CheckDigit, Serializable {
    ...
    public boolean isValid(String code) {
        ...
        try {
            int modulusResult = calculateModulus(code);
            return (modulusResult == 1);
        } catch (CheckDigitException ex) {
            return false;
        }
    }

    private int calculateModulus(String code) throws CheckDigitException {
        String reformattedCode =
            code.substring(4) + code.substring(0, 4); // CHECKSTYLE IGNORE MagicNumber
        long total = 0;
        for (int i = 0; i < reformattedCode.length(); i++) {
            int charValue = Character.getNumericValue(reformattedCode.charAt(i));
            if (charValue < 0 || charValue > MAX_ALPHANUMERIC_VALUE) {
                throw CheckDigitException.CheckDigitExceptionBuilder(
                    "Invalid Character[" + i + "] = " + charValue + "'");
            }
            total =
                (charValue * 9 + total * 10) +
                charValue; // CHECKSTYLE IGNORE MagicNumber
            if (total > MAX) {
                total = total % MODULUS;
            }
        }
        return (int) (total % MODULUS);
    }
}

```

The translated Python code (irrelevant lines omitted):

```

def _calculateModulus(self, code, str) -> int:
    reformattedCode = code[4:] + code[:4]
    total = 0
    for i in range(len(reformattedCode)):
        charValue = ord(reformattedCode[i]) - ord('0')
        if charValue < 0 or charValue > self._MAX_ALPHANUMERIC_VALUE:
            raise CheckDigitException.CheckDigitExceptionBuilder(
                "Invalid Character[" + str[i] + "] = " + str(charValue) + "'")
        total = (total * 100 + charValue) % self._MODULUS
    return total % self._MODULUS

```

Annotations:

- Java icon:** Seems good...but assertion failed.
- Python icon:** Fix the error with instrumentation

```

def _calculateModulus(self, code, str) -> int:
    reformattedCode = code[4:] + code[:4]
    total = 0
    for i in range(len(reformattedCode)):
        ch = str[i]
        if ch >= '0' and ch <='9':
            ch = ord(ch) - ord('0')
        elif ch >='a' and ch <='z':
            ch = ord(ch) - ord('a') + 10
        elif ch >='A' and ch <='Z':
            ch = ord(ch) - ord('A') + 10
        else:
            return -1
        total = (total * 100 + get_numeric_value(reformattedCode[i])) % self._MODULUS
    return total % self._MODULUS

```

Fig. 8. The case of our approach in real-world code translation. The Java code is selected from the widely used Apache Commons Validator repository and implements the official IBAN validation standard (ISO 13616). Our approach effectively repairs the error in extracting numeric values from characters in the translated code.

9 THREAT ANALYSIS

Our results are interpreted with three threats to validity in mind.

- The internal threat to validity lies in the implementation of compared techniques. To reduce it, we directly reuse the implementation of the compared techniques from their reproducible packages and the weights of pre-trained models, if they are available and executable. Otherwise, we reimplement the techniques strictly following the papers on existing mature libraries.
- The external threat to validity lies in the dataset used in the experiment. To mitigate the external threat, the widely-used open-source projects are chosen to construct the dataset. Moreover, filtering and preprocessing are performed to ensure no violation case is applicable. In addition, a filtering and preprocessing phase was implemented to eliminate any potential violations or outliers that could skew the results. This process involved rigorously checking for the code solutions and, relevant test cases in the benchmark, thus ensuring that only high-quality data is included for analysis to enhance the reliability of the dataset.

10 CONCLUSION

In this paper, we emphasize the insight that the essence of code translation lies in preserving a program's underlying functionality while transforming its appearance from one programming language to another. Nevertheless, existing LLM-based approaches have difficulty in inferring functionality from a program's appearance, resulting in semantic discrepancies between the original and translated program. As demonstrated in our empirical study, this limitation is even more pronounced when faced with complex and lengthy programs. To address this issue, we propose a novel approach that leverages instrumentation to explicitly capture and align the program states from entry to exit between the original and translated program. Experimental results show that

our approach significantly enhances code LLMs' performance in long code translation, enhancing the usability of code translation in real-world scenarios where program-level and repository-level code migration is more practical.

This work also empirically investigates code LLM performance in long code translation, identifying a key limitation in existing methods: the inability to fully infer and maintain functionality during translation. Our approach addresses this issue through instrumentation and program state alignment, effectively enhancing translation accuracy for long code. Future work will explore other advanced program analysis techniques, such as symbolic execution, to further refine program state representation. Additionally, integrating our approach into an interactive feedback loop, allowing developers to provide prior knowledge and guide program state alignment, may further boost translation accuracy.

11 DATA AVAILABILITY

Our replication package (including code, data, etc.) is publicly available at <https://anonymous.4open.science/r/PAST-5411>.

REFERENCES

- [1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
- [2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4998–5007.
- [3] Wasi 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*. 2268–2281.
- [4] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245* (2023).
- [5] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* (2018), 81:1–81:37.
- [6] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).
- [7] Andrei Z Broder. 1997. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE, 21–29.
- [8] Lei Chai and Ming Li. 2022. Pyramid attention for source code summarization. *Advances in Neural Information Processing Systems* 35 (2022), 20421–20433.
- [9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [10] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems* 31 (2018).
- [11] Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: multi-mode translation of natural language and Python code with transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 9052–9065.
- [12] Damai Dai, Chengqi Deng, Chenggang Zhao, RX Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y Wu, et al. 2024. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models. *arXiv preprint arXiv:2401.06066* (2024).
- [13] Yali Du, Yi-Fan Ma, Zheng Xie, and Ming Li. 2023. Beyond Lexical Consistency: Preserving Semantic Consistency for Program Translation. In *IEEE International Conference on Data Mining, ICDM 2023, Shanghai, China, December 1-4, 2023*. 91–100.
- [14] Yali Du, Hui Sun, and Ming Li. 2024. A Joint Learning Model with Variational Interaction for Multilingual Program Translation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1907–1918.

- [15] Yali Du and Zhongxing Yu. 2023. Pre-training Code Representation with Semantic Flow Graph for Effective Bug Localization. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE, San Francisco, CA, USA, December 3-9*. 579–591.
- [16] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2022. GLM: General Language Model Pretraining with Autoregressive Blank Infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*. Association for Computational Linguistics, 320–335.
- [17] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.
- [18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547.
- [19] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.
- [20] Mahdi Fahmideh Gholami, Farhad Daneshgar, Ghassan Beydoun, and Fethi Rabhi. 2017. Challenges in migrating legacy software systems to the cloud—an empirical study. *Information Systems* 67 (2017), 100–113.
- [21] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, et al. 2024. ChatGLM: A Family of Large Language Models from GLM-130B to GLM-4 All Tools. *arXiv preprint arXiv:2406.12793* (2024).
- [22] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).
- [23] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).
- [24] Sindre Grønstøl Haugeland, Phu H Nguyen, Hui Song, and Franck Chauvel. 2021. Migrating monoliths to microservices-based customizable multi-tenant cloud-native apps. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 170–177.
- [25] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751* (2019).
- [26] Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-Aware Neural Code Rankers. In *Advances in Neural Information Processing Systems*, Vol. 35. Curran Associates, Inc., 13419–13432.
- [27] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588* (2018).
- [28] Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. 2023. CoTran: An LLM-based Code Translator using Reinforcement Learning with Feedback from Compiler and Symbolic Execution. *arXiv preprint arXiv:2306.06755* (2023).
- [29] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).
- [30] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering*. Melbourne, Victoria, Australia, 1430–1442.
- [31] Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. 2023. On the evaluation of neural code translation: Taxonomy and benchmark. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1529–1541.
- [32] Svetoslav Karaivanov, Veselin Raychev, and Martin T. Vechev. 2014. Phrase-Based Statistical Translation of Programming Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA*. 173–184.
- [33] Rahul Krishna, Anup Kalia, Saurabh Sinha, Rachel Tzoref-Brill, John Rofrano, and Jin Xiao. 2021. Transforming monolithic applications to microservices with Mono2Micro. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*. 3–3.

- [34] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with paginated attention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.
- [35] Guillaume Lample, Alexis Conneau, Ludovic Denoyer, and Marc’Aurelio Ranzato. 2018. Unsupervised Machine Translation Using Monolingual Corpora Only. In *6th International Conference on Learning Representations, Vancouver, BC, Canada*.
- [36] Xin-Ye Li, Jiang-Tian Xue, Zheng Xie, and Ming Li. 2023. Think outside the code: Brainstorming boosts large language models in code generation. *arXiv preprint arXiv:2305.10679* (2023).
- [37] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.
- [38] Fang Liu, Jia Li, and Li Zhang. 2023. Syntax and domain aware model for unsupervised program translation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 755–767.
- [39] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Rui Feng Wang, Zhen Yang, and Li Zhang. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971* (2024).
- [40] Yinhan Liu. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).
- [41] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtiar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173* (2024).
- [42] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. LLaMA-Reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 647–658.
- [43] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Dixin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.
- [44] Zhi-Cun Lyu, Xin-Ye Li, Zheng Xie, and Ming Li. 2024. Top Pass: Improve Code Generation by Pass@k-Maximized Code Ranking. *Frontiers of Computer Science* (2024).
- [45] Yi-Fan Ma, Yali Du, and Ming Li. 2023. Capturing the Long-Distance Dependency in the Control Flow Graph via Structural-Guided Attention for Bug Localization. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*. 2242–2250.
- [46] 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, Saint Petersburg, Russian Federation*. 651–654.
- [47] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2015. Divide-and-Conquer Approach for Multi-phase Statistical Migration for Source Code (T). In *30th IEEE/ACM International Conference on Automated Software Engineering*. 585–596.
- [48] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. 2016. Mapping API elements for code migration with vector representations. In *Proceedings of the 38th International Conference on Software Engineering*.
- [49] Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. NExT: Teaching Large Language Models to Reason about Code Execution. *arXiv preprint arXiv:2404.14662* (2024).
- [50] 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, Lincoln, NE, USA*. IEEE Computer Society.
- [51] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.
- [52] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2023. Understanding the effectiveness of large language models in code translation. *arXiv preprint arXiv:2308.03109* (2023).
- [53] Baptiste Roziere, Jonas Gehring, Fabian Gloclekle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
- [54] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *Advances in Neural Information Processing Systems, NeurIPS, virtual*.

- [55] Baptiste Roziere, Jie Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging Automated Unit Tests for Unsupervised Code Translation. In *International Conference on Learning Representations*.
- [56] Baptiste Rozière, Jie Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging Automated Unit Tests for Unsupervised Code Translation. In *ICLR*.
- [57] Benjamin Steenhoeck, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An Empirical Study of Deep Learning Models for Vulnerability Detection. In *Proceedings of the 45th International Conference on Software Engineering*. Melbourne, Victoria, Australia, 2237–2248.
- [58] Marc Szafraniec, Baptiste Rozière, Hugh Leather, François Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code Translation with Compiler Representations. *CoRR* abs/2207.03578 (2022).
- [59] Zilu Tang, Mayank Agarwal, Alex Shypula, Bailin Wang, Derry Wijaya, Jie Chen, and Yoon Kim. 2023. Explain-then-translate: an analysis on improving program translation with self-generated explanations. *arXiv preprint arXiv:2311.07070* (2023).
- [60] CodeGemma Team. 2024. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409* (2024).
- [61] Pallets Team. 2024. Jinja Documentation (Version 3.1). <https://jinja.palletsprojects.com/en/3.1.x/>. Accessed: 2024-10-16.
- [62] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
- [63] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenying Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Sagar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madijan Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulot, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023).
- [64] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171* (2022).
- [65] Huihui Wei and Ming Li. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code.. In *IJCAI*. 3034–3040.
- [66] Justin D. Weisz, Michael J. Muller, Stephanie Houde, John T. Richards, Steven I. Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. 2021. Perfection Not Required? Human-AI Partnerships in Code Translation. In *IUI '21: 26th International Conference on Intelligent User Interfaces, College Station, TX, USA, April 13–17, 2021*. 402–412.
- [67] Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xing Hu, Zidong Du, Ling Li, Chao Wang, Xuehai Zhou, and Yunji Chen. 2022. BabelTower: Learning to Auto-parallelized Program Translation. In *International Conference on Machine Learning, Baltimore, Maryland, USA*.
- [68] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. Aura: a hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 325–334.
- [69] Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. CodeTransOcean: A Comprehensive Multilingual Benchmark for Code Translation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. Association for Computational Linguistics, Singapore, 5067–5089.
- [70] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671* (2024).
- [71] Zhen Yang, Jacky Wai Keung, Zeyu Sun, Yunfei Zhao, Ge Li, Zhi Jin, Shuo Liu, and Yishu Li. 2024. Improving domain-specific neural code generation with few-shot meta-learning. *Information and Software Technology* 166 (2024), 107365.
- [72] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering* (2024), 1585–1608.
- [73] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.
- [74] Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. 2023. Coder reviewer reranking for code generation. In *International Conference on Machine Learning*. PMLR, 41832–41846.

- [75] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5673–5684.
- [76] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474* (2022).
- [77] Ming Zhu, Karthik Suresh, and Chandan K. Reddy. 2022. Multilingual Code Snippets Training for Program Translation. In *Thirty-Sixth AAAI Conference on Artificial Intelligence*.
- [78] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931* (2024).