Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code

Rangeet Pan*
rangeet.pan@ibm.com
IBM Research
Yorktown Heights, NY, USA

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

Rahul Krishna
rkrsn@ibm.com
IBM Research
Yorktown Heights, NY, USA

Divya Sankar
divya.sankar@ibm.com
IBM Research
Yorktown Heights, NY, USA

Lambert Pougeum Wassi
lambert.pouguem.wassi@ibm.com
IBM Research
Yorktown Heights, NY, USA

Michele Merler
mimerler@us.ibm.com
IBM Research
Yorktown Heights, NY, USA

Boris Sobolev
bsobolev@ibm.com
IBM Research
Yorktown Heights, NY, USA

Raju Pavuluri
pavuluri@us.ibm.com
IBM Research
Yorktown Heights, NY, USA

Saurabh Sinha
sinhas@us.ibm.com
IBM Research
Yorktown Heights, NY, USA

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

ABSTRACT

Code translation aims to convert source code from one programming language (PL) to another. Given the promising abilities of large language models (LLMs) in code synthesis, researchers are exploring their potential to automate code translation. The prerequisite for advancing the state of LLM-based code translation is to understand their promises and limitations over existing techniques. To that end, we present a large-scale empirical study to investigate the ability of general LLMs and code LLMs for code translation across pairs of different languages, including C, C++, Go, Java, and Python. Our study, which involves the translation of 1,700 code samples from three benchmarks and two real-world projects, reveals that LLMs are yet to be reliably used to automate code translation—with correct translations ranging from 2.1% to 47.3% for the studied LLMs. Further manual investigation of *unsuccessful* translations identifies 15 categories of translation bugs. We also compare LLM-based code translation with traditional non-LLM-based approaches. Our analysis shows that these two classes of techniques have their own strengths and weaknesses. Finally, insights from our study suggest that providing more context to LLMs during translation can help them produce better results. To that end, we propose a prompt-crafting approach based on the symptoms of erroneous translations; this improves the performance of LLM-based code translation by 5.5% on average. Our study is the first of its kind, in terms of scale and breadth, that provides insights into the current limitations of LLMs in code translation and opportunities for improving them. Our dataset—consisting of 1,700 code samples in five PLs with 10K+ tests, 43K+ translated code, 1,725 manually labeled bugs, and 1,365 bug-fix pairs—can help drive research in this area.

*Both authors contributed equally to this research.
†Work has been done when Ali was an intern at IBM Research.

CCS CONCEPTS

• **General and reference → Empirical studies**; • **Computing methodologies → Neural networks**.

KEYWORDS

Code Translation, Bug Taxonomy, LLMs

ACM Reference Format:
Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pougeum Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Code translation, source-to-source compilation, or transpilation, entails transforming a piece of code from one programming language (PL) to another, while preserving the original functionality. Code translation has many use cases, such as modernizing enterprise applications [30, 46, 49, 62, 68], migrating legacy software in proprietary PLs to cloud-native applications implemented in general-purpose PLs [26, 34, 36, 48, 59, 61, 89], and facilitating the training of models for better code synthesis [28, 37, 69, 70]. Translating the software/code to a modern PL can significantly reduce maintenance effort, improve overall reliability, and boost non-functional properties such as security and performance [1–5, 74].

Due to the importance and benefits of code translation, several techniques have been developed to automate reliable translation between different PLs [8, 9, 15, 29, 38, 53, 57, 65, 66, 73, 75, 79–81], including those leveraging *Large Language Models* (LLMs) for code translation [35, 43, 65, 66, 72, 79]. Although prior research has shown the potential of using LLMs for code translation, there is a dearth of research on understanding and cataloging their limitations for this task. This is an important undertaking because code translation is a complex task that requires LLMs to understand code syntax (to generate syntactically correct code) and semantics (to preserve functionality during translation) simultaneously. However, research has shown that without providing adequate context to LLMs via prompt crafting, they may only serve as "next code token" predictors, without understanding the overall task [22, 44, 82, 92].

In this work, we perform a large-scale empirical study to understand the promises and limitations of LLM-based code translation, and compare them with existing non-LLM-based translation approaches. We also perform a preliminary investigation of how providing more context about incorrect translations improves the results. Our study answers the following research questions:

RQ1: Effectiveness in Code Translation (§3). (*RQ1.1*) How do state-of-the-art general and code LLMs perform in code translation? (*RQ1.2*) What are the outcomes of unsuccessful translations?
RQ2: LLM-Based Translation Bugs (§4). (*RQ2.1*) What are the different types of underlying root causes (translation bugs) for unsuccessful translation? (*RQ2.2*) How prevalent are these bugs in unsuccessful translations? (*RQ2.3*) How do translation bugs in *real-world projects* differ from those in *crafted benchmarks*?
RQ3: Comparison with Alternative Approaches (§5). How do state-of-the-art non-LLM-based techniques perform in code translation and what types of translation bugs do they introduce?
RQ4: Mitigating Translation Bugs (§6). To what extent do the proposed *prompt-crafting techniques* resolve translation bugs?

To investigate the RQs, we collected 1,700 executable code samples from *three* well-known datasets (CodeNet [63], Avatar [23], and EvalPlus [54]) and *two* open-source projects (Apache Commons CLI [6] and Python Click [10]), covering *five* PLs (C, C++, Go, Java, and Python). To perform translation, we selected *seven* LLMs: GPT-4 [60], which has shown promising performance in various tasks [12], *three* open-source LLMs from the Hugging Face Open LLM Leaderboard [13] (Llama 2 [16], TheBloke-Vicuna [20], and TheBloke-Airoboros [19]), and *three* recent code LLMs (StarCoder [51], CodeGeeX [91], and CodeGen [58]).

We performed 43,379 translations across all LLMs, measuring translation success against the tests provided with the code samples. This produced 11.94% successful translations on average (median 5.3%), with GPT-4 (47.3% success rate) and StarCoder (14.5% success rate) being the best-performing models (details in §3.1). On real-world projects, the LLMs were largely ineffective, with success rates of 8.1% for GPT-4 and 0% for the rest of the models.

We also conducted a systematic study to understand the root causes of unsuccessful translations and create a taxonomy of translation bugs. The process involved eight human labelers and took 630 person-hours in total, focusing on 1,725 buggy translations by GPT-4. It was conducted in two phases: in Phase 1, a draft taxonomy was created from buggy translations for one language pair; in Phase 2, the taxonomy was used for other language pairs and extended, if needed, to label buggy translations (details in §4). The resulting bug taxonomy is structured into 15 categories and five groups (details in §4.1). Some notable findings are: (1) identifying suitable data type in the target PL that preserves the source behavior is challenging, (2) identifying equivalent APIs in the target language or implementing the API functionality can introduce bugs, and (3) replacing language-specific features, such as method overloading and annotations, can be challenging, especially in real-world projects. Another important dimension of this study is comparing LLM-based translation with existing non-LLM-based techniques, namely, CxGo [8], C2Rust [9], and JavaToCsharp [15].[1] The comparison shows that LLM- and non-LLM-based translation techniques provide different and unique advantages, suggesting that an ultimate solution for code translation should combine both techniques (details in §5).

Our study reveals that, often, providing only the source code may be insufficient for achieving correct code translation. To that end (and also motivated by recent research on LLM-based bug repair [45, 84]), we propose an iterative prompting approach that incorporates additional informative context in prompts corresponding to the previously unsuccessful translation, including the code, stack trace, error message from failing execution, and/or test input and expected output from failing test cases. Our results show prompt crafting increases the success rate by 5.5%, on average, across the studied LLMs, with the largest improvement, of 12%, occurring for GPT-4 (details in §6). Although these results are encouraging, they indicate considerable scope for improvement, likely through a combination of program analysis techniques and LLMs.

To our knowledge, we are the first to (1) provide a systematic bug taxonomy facilitating a deeper understanding of error modalities in LLM-based code translation, (2) study translation of real-world projects, (3) investigate the effectiveness of prompt crafting in mitigating translation bugs, and (4) compare non-LLM and LLM-based translation approaches. Our key contributions are:

- **A comprehensive evaluation of LLM-based code translation.** We perform a large-scale evaluation of the code translation using multiple general and code LLMs. We consider the recently released LLMs, and our evaluation includes real-world projects in addition to three crafted benchmarks.
- **A taxonomy of translation bugs.** Our study offers the first taxonomy of bugs introduced by LLMs during code translation. We also compare the nature of these bugs in LLM and non-LLM-based approaches to understand their strengths and weaknesses.

[1]We also considered other approaches (mppSMT [57], Tree2Tree [29], and Sharpen [18]) but could not compare against them due to lack of tool/artifact availability (§5).

Table 1: Overview of subject LLMs. TB: TheBloke.

Modality	**Code**			**Text**			
Models	CodeGen	CodeGeeX	StarCoder	GPT-4	Llama 2	TB-Airoboros	TB-Vicuna
Size	16B	13B	15.5B	-	13B	13B	13B
Context Window	2048	2048	2048	8192	4096	2048	2048
Release Date	May'23	Mar'23	May'23	Mar'23	Jul'23	May'23	May'23

- **Prompt crafting to enhance code translation.** A set of heuristics for prompt crafting that provides proper contexts to LLMs to improve their effectiveness in code translation.
- **Artifacts.** Our artifacts, including labeled bug instances and their outcomes, implementation of prompt crafting, and automation scripts for assessing LLM performance, are publicly available [7].

2 EMPIRICAL SETUP

Subject LLM Selection. General LLMs are pre-trained on textual data, including natural language and code, and can be used for a variety of tasks. In contrast, code LLMs are specifically pre-trained to automate code-related tasks. Due to the empirical nature of this work, we were interested in assessing the effectiveness of both LLM categories in code translation. For code LLMs, we selected the top three models released recently (in 2023), namely CodeGen [58], StarCoder [51], and CodeGeeX [91]. For general LLMs, we selected the top three models with size 20B parameters or less from the Hugging Face Open LLM Leaderboard [13].[2] The constraint on the number of parameters was imposed by our computing resources, resulting in the selection of Llama 2 [16], TheBloke-Airoboros [19], and TheBloke-Vicuna [20]. We also included GPT-4 [60] in our study. Table 1 summarizes characteristics of the selected LLMs.

Subject PLs Selection. We used the following criteria to select the subject PLs: (1) popularity of the language based on the TIOBE index [21], (2) inclusion of different programming paradigms, e.g., procedural, object-oriented, and functional, and (3) availability of high-quality datasets in the given PL. To make the manual effort involved in taxonomy construction manageable, we selected five PLs that met the inclusion criteria—C, C++, Go, Java, and Python.

Dataset Collection and Pre-Processing. To ensure the comprehensiveness of our findings and claims in understanding the nature of LLM translation bugs, we were interested in datasets used in prior studies as well as real-world projects. The former consists of small programs, likely to be less challenging for LLMs to translate, and the latter assesses the complexity of LLM translation in real-world settings.

The first six columns of Table 2 present the selected datasets and statistics about them (more information in the artifact website [7]). These datasets are accompanied by test cases to validate code translation. For CodeNet and Avatar, the tests are input data and expected output, while EvalPlus and real-world projects have unit tests (JUnit and pytest). For EvalPlus, we manually translated and verified the corresponding pytests to JUnit tests. The translation of real-world projects never reached test execution, as it produced syntactically incorrect code (more discussion in §3).

For real-world projects, we focused on Java and Python, the most popular languages among our subject PLs. Our goal was to translate reasonably complex and well-maintained software exclusively written in Java or Python. To that end, we selected projects available in both PLs providing APIs for command-line processing and selected Apache Commons CLI [6] (Java) and Click [10] (Python). To fit the source language code into the limited LLM context window, we broke them down into classes and files and removed all comments.

Compute Resources. To perform inference on all subject LLMs, we used 16 A100 80GB memory GPUs. For evaluating the generated translations, we used Python 3.10, g++ 11, GCC Clang 14.0, Java 11, and Go 1.20 for Python, C++, C, Java, and Go, respectively.

3 LLM-BASED CODE TRANSLATION

We prompted each subject LLMs with 6,197 translation problems corresponding to 31 translation pairs shown in Table 2, i.e., 20 pairs from CodeNet, eight pairs from Avatar, and one pair each for EvalPlus, Commons CLI, and Click. Through RQ1, we evaluate the effectiveness of LLMs in code translation (RQ1.1) and the outcomes of incorrect translations (RQ1.2).

3.1 Effectiveness of LLMs in Code Translation

We refer to the LLM prompting in this experiment as *vanilla prompting*, where each prompt contains four pieces of information: (1) instructions in natural language to perform the translation task, (2) source language (`$SOURCE_LANG`), (3) target language (`$TARGET_LANG`), and (4) the code to be translated (`$SOURCE_CODE`). We followed the templates similar to those we found in the artifacts, papers, or technical reports associated with each model. Figure 1 shows the three templates used for vanilla prompting of our subject LLMs. Our prompt template for CodeGeeX slightly differs from what is used in their paper [91]. Specifically, their prompt template includes imports, class declaration, and method signature of the translation [11]. However, this is an *unrealistic approach* because such ground truth does not exist and requires human involvement for each translation. Moreover, the code to be translated (from real-world projects or crafted benchmarks) often contain several methods, making it impossible to use the same template.

We consider a translation successful if it compiles, passes runtime checks, and existing tests pass on the translated code. We do not consider static evaluation metrics such as exact match, syntax match, dataflow match [64], CodeBLEU [64], and CrystalBLEU [31] because our goal is to validate (compile and execute) the translations. Static metrics can also be misleading in code synthesis [27]—i.e., LLMs may achieve reasonably high numbers for these metrics, but generate code that cannot be executed due to compilation or runtime errors [23, 27]. The last seven columns of Table 2 show the detailed results of vanilla prompting of subject LLMs for code translation. We next discuss our key observations.

- Except for GPT-4 and StarCoder, all other models performed poorly. The biggest surprise here is CodeGeeX, a model trained explicitly for code translation. We believe this result is because, as mentioned, we excluded information about the translated code (imports, class declaration, and method signature) in the prompt. Such information is typically not available and non-trivial to compute. (To check the correctness of our results, we repeated their experiments with their template and ours, which resulted in the pass@1 dropping from 25.6% to 0.02% on their dataset, HumanEval-X.)
- There is a strong correlation between the average number of tests per translation sample and unsuccessful translation (correlation

[2]The Open LLM Leaderboard ranking is quite dynamic, and our selection is drawn from the ranking at the time of our experimentation.

Table 2: Performance of subject LLMs in translating code from different studied datasets. The best performance by general and code LLMs are highlighted in teal and violet, respectively. The final performance is computed over the average of each dataset.

Dataset	Source Language	Source Samples	#Tests	Target Language	#Translations	% Successful Translations						
						CodeGen	CodeGeeX	StarCoder	GPT-4	Llama 2	TB-Airoboros	TB-Vicuna
CodeNet [63]	C	200	200	C++, Go, Java, Python	800	23.4%	14.9%	**42.0%**	**83.0%**	14.9%	18.8%	4.4%
	C++	200	200	C, Go, Java, Python	800	14.0%	3.6%	**39.1%**	**80.0%**	9.5%	8.3%	3.4%
	Go	200	200	C, C++, Java, Python	800	14.3%	5.9%	**42.0%**	**85.5%**	16.9%	6.6%	0.9%
	Java	200	200	C, C++, Go, Python	800	21.3%	10.3%	**30.3%**	**81.3%**	13.9%	6.5%	0.1%
	Python	200	200	C, C++, Go, Java	800	17.5%	7.3%	**33.3%**	**79.9%**	11.0%	6.5%	1.0%
Total/Average (CodeNet)	-	1,000	1,000	-	4,000	18.1%	8.4%	**37.3%**	**82.0%**	13.2%	9.3%	2.0%
Avatar [23]	Java	249	6,255	C, C++, Go, Python	996	8.1%	1.8%	**11.9%**	**70.8%**	1.8%	5.1%	0.0%
	Python	250		C, C++, Go, Java	1, 000	3.8%	1.6%	**14.2%**	**52.2%**	4.7%	0.9%	0.9%
Total/Average (Avatar)	-	499	6,255	-	1,996	5.9%	1.7%	**13.0%**	**61.5%**	3.2%	3.0%	0.4%
EvalPlus [54]	Python	164	2,682	Java	164	16.5%	3.7%	**22.0%**	**79.3%**	1.2%	14.0%	7.9%
Commons CLI [6]	Java	22	310	Python	22	0.0%	0.0%	0.0%	**13.6%**	0.0%	0.0%	0.0%
Click [10]	Python	15	611	Java	15	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Total/Average (All)	-	1,700	10,858	-	6,197	8.1%	2.8%	**14.5%**	**47.3%**	3.5%	5.3%	2.1%

Table 3: Breakdown of the unsuccessful translations produced by subject LLMs based on outcome. All values are in %.

Source Language	C				C++				Go				Java				Python				Total
Target language	C++	Go	Java	Python	C	Go	Java	Python	C	C++	Java	Python	C	C++	Go	Python	C	C++	Go	Java	
Compilation Errors	68.9	93.5	76.4	56.9	93.2	94.6	77.0	61.6	86.7	83.3	82.4	55.9	82.4	78.4	96.6	57.4	79.9	73.4	86.0	72.4	**77.8**
Runtime Errors	9.4	2.3	10.7	21.9	0.1	1.2	11.2	22.9	0.2	0.2	12.7	19.3	1.2	0.4	0.8	27.1	0.4	0.4	10.0	14.8	**8.4**
Functional Errors	20.5	3.7	13.0	20.6	6.7	4.1	11.7	15.1	12.9	16.3	4.7	24.6	15.8	19.9	2.5	15.1	19.0	24.8	3.9	12.5	**13.4**
Non-terminating Execution	1.3	0.4	0.0	0.5	0.0	0.2	0.1	0.4	0.2	0.2	0.2	0.2	0.7	1.3	0.1	0.3	0.8	1.3	0.1	0.3	**0.4**

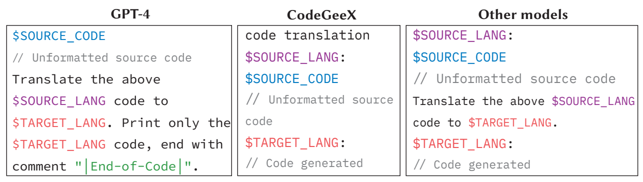

GPT-4
```
$SOURCE_CODE
// Unformatted source code
Translate the above
$SOURCE_LANG code to
$TARGET_LANG. Print only the
$TARGET_LANG code, end with
comment "|End-of-Code|".
```
CodeGeeX
```
code translation
$SOURCE_LANG:
$SOURCE_CODE
// Unformatted source
code
$TARGET_LANG:
// Code generated
```
Other models
```
$SOURCE_LANG:
$SOURCE_CODE
// Unformatted source code
Translate the above $SOURCE_LANG
code to $TARGET_LANG.
$TARGET_LANG:
// Code generated
```

Figure 1: Vanilla prompting templates.

coefficient *r* ranging from 0.64 to 0.85 for all models). That is, the more rigorous the existing test suite, the better it can evaluate if a translation preserves functionality.

- There is no consistent pattern between unsuccessful translations and source/target language, but translating *to* Go results in more compilation errors due to its strict syntax constraints, e.g., forbidding unused variables or imports.
- The LLMs fail to translate real-world projects. This is mainly because crafted benchmark programs are simpler, without complex dependencies or use of language features, e.g., annotations, inheritance, etc. Moreover, in a real-world setting, translating files/methods in isolation, even if successful, may fail at the project level. That said, further manual investigation showed that for the Commons CLI, three out of 22 translated files could be compiled using `py_compile` [17]. These simple classes consist of (1) an exception class with only one method, (2) an interface with two method declarations, and (3) a utility class with two simple methods.

3.2 Outcome of Unsuccessful Translations

The previous research question shows that most of the subject LLMs are yet to achieve a reasonable performance for code translation, even on crafted benchmarks, let alone real-world projects. At the next step, we were interested to understand if this is due to a lack of understanding of code syntax or semantics by LLM. To do this, we classify unsuccessful translations based on their error outcome: (1) *Compilation Error*, where translated code cannot be compiled, (2) *Runtime Error*, where translated code compiles but fails at runtime with an exception, (3) *Functional Error*, where the translated code compiles and executes successfully but results in test failure, and (4) *Non-terminating Execution*, where the translated code compiles and executes, but does not terminate (encountering an infinite loop or waiting on user input).

Figure 2 and Table 3 show the results of this experiment for each model—accumulated for all subject PLs—and for each subject PL—accumulated for all subject models, respectively. From these results, we observe that most unsuccessful translations result in compilation errors (77.8%), meaning both general and code LLMs have difficulty understanding code syntax. Further breakdown of the results per PLs shows that Go and C++ have comparatively stricter syntax, while it is easier for LLMs to generate syntactically correct Python code.[3] The next most common effect of unsuccessful translation is a functional error (13.4%), demonstrating that often translated code does not preserve the behavior of the source program.

4 LLM-BASED TRANSLATION BUGS

To understand the nature of translation bugs, we performed a deep analysis by manually investigating the root cause of unsuccessful translations. Through the following three research questions, we introduce our comprehensive taxonomy of translation bugs (RQ2.1), investigate the prevalence and distribution of each bug category across unsuccessful translations (RQ2.2), and discuss the peculiar characteristics of bugs in real-world projects (RQ2.3).

4.1 Taxonomy of Translation Bugs

Our initial investigation showed that GPT-4 is the best-performing model and compared to others, its translations exhibit a variety of quality bugs worth investigating. To manage the manual effort for understanding and labeling bugs for building the taxonomy, we focused on 1,725 unsuccessful translations from GPT-4.

[3]We used `py_compile` [17] to check syntax-related bugs in Python.

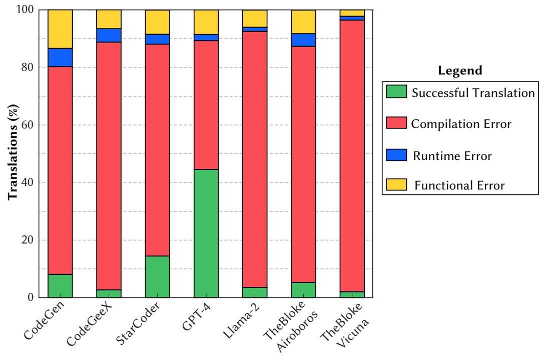


Figure 2: Outcome of code translations using subject LLMs.

4.1.1 Methodology. The manual construction of the taxonomy involved eight human labelers, who are researchers or software engineers in the industry, and involved unsuccessful translations from all 31 translation pairs in Table 2. We built the taxonomy in two phases. In the first phase, we used the CodeNet Java-to-Python translations. Each labeler created a taxonomy independently by examining the unsuccessful translations. Then, we combined the individual taxonomies to create a consolidated taxonomy, which served as the initial taxonomy for phase 2. In the second phase, each of the remaining 30 translation pairs was examined by two labelers to assign bug categories to unsuccessful translations. Whenever a new category came up (i.e., a bug could not be covered by the existing categories in the taxonomy), the entire team met to discuss the new category, add it to the taxonomy, and re-label the affected bugs. After completing their labeling tasks, the two labelers assigned to a translation pair met to discuss their labeling, resolve discrepancies, and create the final labeling for the translation pair. The entire exercise took about 630 person-hours and produced a taxonomy with 15 bug categories organized into five groups ("model specific constraints" group does not have any sub-category). In the rest of this section, we discuss the taxonomy groups and bug categories together with illustrative examples.

4.1.2 Syntactic and semantic differences between the source and the target languages. There are five bug categories in this group that relate to the failure of an LLM in appropriately handling the syntactic or semantic differences between PLs.

A1: Violating target language requirements. Each PL has its own set of rules that the code must adhere to. For example, in Java, any executable code must be wrapped in a method within a class, whereas in other PLs, e.g., Python, this is not a requirement. For another example, unused imports result in compilation errors in Go, but not in other languages.

A2: Duplicating source syntax to the target language. LLMs often copy the source PL syntax even if they are not available in the target PL. In an unsuccessful translation below from C++ to Go, the subject LLM does not replace `atan2l` API (which is specific to C++ and does not exist in Go) with an appropriate equivalent.

```
const ld PI = atan2l(0, -1); # Original C++ code
PI = atan2l(0, -1) # Incorrect Go code
```

A3: Mismatch of API behaviors in the source and target. Library APIs are frequently used in programs in any PL. During translation, API calls in the source need to be either mapped to equivalent API calls in the target PL or implemented from scratch. In the former case, we noted that LLMs often map source APIs incorrectly to the target PL. The following code fragment illustrates such an example where the Java `String.substring()` API is incorrectly mapped to the Go `strings.IndexByte()` API method.

```
S.substring(i, i + 1) # Original Java code (returns String)
strings.IndexByte(S, i) # Incorrect Go code (returns Int64)
```

A4: Incorrect use of operators. The supported operators and their syntax can vary among PLs. For example, `//` in Python represents floor division, for which Java has no corresponding operator. To achieve that behavior in Java, a division must be followed by a call to `Math.floor()`. LLMs can fail in translating such cases.

```
i = i // 10; # Original Python code (floor division)
i = i / 10; # Incorrect Java code (division)
```

4.1.3 Dependency and logic bugs in the translated code. These bugs pertain to incorrect dependencies and logic of translated code.

B1: Missing library imports. Import statements are used to load libraries and/or application classes/modules used in code. In several unsuccessful translations, we found that the translated code had missing or incorrect imports.

B2: Missing definition. LLMs can omit definitions/implementation of data types, methods, etc. In the unsuccessful translation below, the original C++ code, `main()` calls method `solve()` that is implemented in the same source file. However, after translation to Go, although the call to `solve()` remains, its definition is removed.

```
void solve(){...} # Original C++ code
signed main(){...solve();} # Incorrect Go code
```

B3: Incorrect loop and conditional statements. This category covers bugs in translating loops and conditional statements. The following code illustrates the incorrect translation of a Java loop to Python array slicing, resulting in an off-by-one error in the translated code, where the sum excludes the value of `x[200010-k-1]`.

```
for(int i = 0; i <= 200010 - k - 1; i++) ans += x[i]; # Java code
ans = sum(x[:200010 - k - 1]) # Incorrect Python code
```

B4: Inclusion of logic not in source code. LLMs can generate code that is unrelated to any logic in the source program, thereby, causing the translated code to diverge from the source behavior. In the following example, `max` is initialized to `-1` in the source. However, after translation, it is initialized to a different value; the LLM thus adds logic that does not exist in the source program.

```
int max = -1; # Original C code
max_h = max(h) # Incorrect Python code
```

B5: Removal of logic in the source code. LLMs often fail to translate part of the source program correctly. For example, in several cases lines such as `#define MAX 101` in C are removed.

B6: Mismatch of behavior after replacing API call. These bugs occur when a source API call is translated to custom logic instead of an equivalent API in the target PL (A3 bugs).

```
return Collections.unmodifiableList(requiredOpts); # Java code
return self.required_opts # Incorrect Python code
```

The example above from Commons CLI shows an incorrect translation of Java `Collections.unmodifiableList()` that returns an immutable list to Python, where the returned list is mutable.

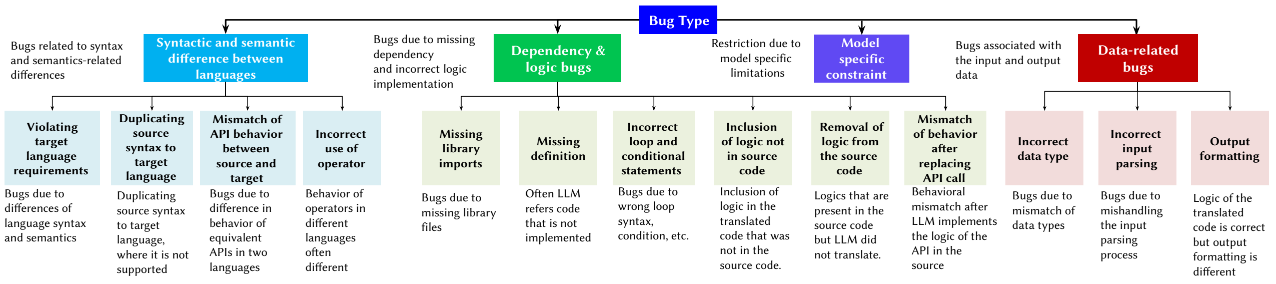

Bug Type
Syntactic and semantic difference between languages
Bugs related to syntax and semantics-related differences
Dependency & logic bugs
Bugs due to missing dependency and incorrect logic implementation
Model specific constraint
Restriction due to model specific limitations
Data-related bugs
Bugs associated with the input and output data
Violating target language requirements
Bugs due to differences of language syntax and semantics
Duplicating source syntax to target language
Duplicating source syntax to target language, where it is not supported
Mismatch of API behavior between source and target
Bugs due to difference in behavior of equivalent APIs in two languages
Incorrect use of operator
Behavior of operators in different languages often different
Missing library imports
Bugs due to missing library files
Missing definition
Often LLM refers code that is not implemented
Incorrect loop and conditional statements
Bugs due to wrong loop syntax, condition, etc.
Inclusion of logic not in source code
Inclusion of logic in the translated code that was not in the source code.
Removal of logic from the source code
Logics that are present in the source code but LLM did not translate.
Mismatch of behavior after replacing API call
Behavioral mismatch after LLM implements the logic of the API in the source
Incorrect data type
Bugs due to mismatch of data types
Incorrect input parsing
Bugs due to mishandling the input parsing process
Output formatting
Logic of the translated code is correct but output formatting is different

Figure 3: Taxonomy of bugs introduced while translating code using LLM. The “other” category/group is not shown here.

4.1.4 *Data-related Bugs.*

Data plays an integral role in the code. We found various bugs caused by wrong assumptions about the input data, mismatch of data types, etc.

C1: Incorrect data type. This category of bugs pertains to incorrect types assigned to variables. The following example, taken from the translation of Commons CLI to Python, illustrates an incorrect type assignment to a field.

```
public static Class<File[]> FILES_VALUE=File[].class; # Java code
FILES_VALUE = List[os.path] # Incorrect Python code
```

C2: Incorrect input parsing. Programs that read data from the input stream (i.e., `stdin` or other sources expect the data to be in a specific format. Such programs are often translated incorrectly, as illustrated by the following Python-to-C translation. The second input line contains three integer values all of which are read into an array in the Python code (line 6), whereas the C code reads only two values and assigns 0 for the third value (lines 2–4). This causes the wrong result to be computed in line 6.

```
0   ----- Input -----
1 2
2 3 5 2
3 4 5
4 ------ Source Code ------
5 N = int(input())
6 A = [.. input().split()]
7 ...
8     d = min(A[i+1], B[i])
```

```
0   ----- Translated Code -----
1 int A[N+1], B[N], N;
2 for(int i=0; i<N; i++) {
3     scanf("%d", &A[i]);}
4 A[N] = 0;
5 . . .
6     d = A[i+1] < B[i] ? A[i+1] :
          B[i];
7 ...
```

C3: Output formatting. We found that often, even if the translated code logic is correct, the output is formatted differently. In the following example, the source Java code prints ‘H’ followed immediately by an integer value, whereas the translated Python code prints a space between ‘H’ and the integer value.

```
System.out.print("H"); System.out.println(Y - 1988); # Java code
print("H", Y - 1988) # Incorrect Python code
```

4.1.5 *D: Model-Specific Bugs.*

Some bugs are specific to the design of the LLMs used. For instance, having natural language in-between the code, exceeding token size, etc., causing compilation errors or no output to be generated. We also observed a group of unsuccessful translations—which we refer to as *E: Others*—related to our experiment setup (e.g., memory issues). Given that they do not represent LLM-introduced bugs, we do not include them in the taxonomy.

4.2 Prevalence of LLM-based Translation Bugs

We now present the results for RQ2.2, showing the prevalence of bugs in different categories of our taxonomy Among the 6197 attempted translations over 31 language pairs (Table 2), there were 1558 translation failures. We manually checked and labeled these failures to identify 1725 bugs. In many cases, an unsuccessful translation has multiple bugs that belong to different categories; in these cases, the translation gets multiple labels. Table 4 presents detailed results on the prevalence of translation bugs. In this section, we delve deeper into the characteristics and prevalence of these bugs.

Finding 1: More than one-third (33.5%) of the translation bugs are data-related bugs.

As the data for bug group C in Table 4 show, a large proportion of the LLM-introduced bugs is related to data types, parsing input data, and output formatting issues, together accounting for 33.5% of all bugs. These bugs are particularly prevalent for Python to C, C++, and Go translations, constituting over 55%, 64%, and 67% of the bugs, respectively, in those translations. Within data-related bugs, our manual investigation found several unique patterns.

Finding 1a: Among the data-related bugs, most (54% of the data-related bugs and 18.1% of the total bugs) are due to incorrect parsing of inputs.

As discussed in §4.1.4, programs that take external inputs contain input-parsing logic, assuming the data to adhere to certain formats, and LLMs often make mistakes while translating this logic; the C2 bug category example in §4.1.4 illustrates this. A major reason for the prevalence of this bug is that two of our datasets (CodeNet and Avatar) consist of programs that read data from the input stream. For EvalPlus and the real-world projects, the occurrence of this bug is lesser (one for EvalPlus and none for the real-world projects).

Finding 1b: Choosing the correct data type in the target PL is a crucial step that accounts for 34.3% of all data-related bugs and 11.5% of all bugs.

Assignment of correct data types in the translated code (11.5% of translation bug) is a major challenge. These bugs occur due to incorrect choice of the data type, differences between behaviors of equivalent data types across PLs, and differences in type systems of the source and target PLs. The example for the C1 bug category in §4.1.4 shows an instance of wrong choice of data type in the target PL, where the Java type `Class<File[]>` is converted to list of path objects in Python. To illustrate an example of equivalent types with different behaviors in source and target PLs, consider the code fragments shown below, where the function `mean_a_d()` in the Python code (left) takes a list of `float` as input. The test case for the function (line 2) uses a large number as test data. The translated code, shown on the right, looks correct and maps Python `float` to the Java `Float` type. However, the equivalent Java test case (line 2 on right) fails because of a fundamental difference between Python `float` and Java `Float`: the former uses 64-bit precision whereas the

Table 4: Types of bugs introduced during code translation by LLM and their occurrences. This table consists all the subject datasets including real-world projects. All values are in %.

Source Language	C				C++				Go				Java				Python				Total
Target Language	C++	Go	Java	Py	C	Go	Java	Py	C	C++	Java	Py	C	C++	Go	Py	C	C++	Go	Java	
A1: Violating target language requirements	25.9	8.3	20.4	14.5	23.1	37.5	2.7	5.8	78.2	58.5	55.8	41.4	57.7	23.5	3.6	11.2	20.0	11.3	10.0	4.6	24.3
A2: Duplicating source syntax to the target language	18.5	2.8	0.9	4.6	0.0	4.2	6.8	1.2	0.0	3.1	0.6	1.8	0.0	5.9	3.6	2.9	0.0	0.0	2.5	1.1	2.4
A3: Mismatch of API behaviors in the source and target	0.0	11.1	4.4	0.0	0.0	0.0	5.5	2.9	1.8	0.0	3.5	0.5	0.0	0.0	0.0	1.0	0.0	0.0	2.5	16.1	3.3
A4: Incorrect use of operators	0.0	0.0	0.0	1.2	0.0	0.0	1.4	0.0	0.0	0.0	0.0	0.0	0.0	2.9	3.6	2.4	2.2	0.0	0.0	0.0	0.6
A: Syntactic and semantic differences between languages	44.4	22.2	25.7	20.2	23.1	41.7	16.4	9.9	80.0	61.5	59.9	43.6	57.7	32.4	10.7	17.6	22.2	11.3	15.0	21.8	30.5
B1: Missing library imports	0.0	8.3	3.5	7.5	0.0	0.0	5.5	32.0	0.0	1.5	1.2	2.7	7.7	0.0	0.0	22.4	0.0	0.0	0.0	8.6	8.6
B2: Missing definition	11.1	13.9	2.7	3.5	0.0	4.2	4.1	5.8	0.0	3.1	0.0	0.5	7.7	14.7	0.0	0.0	0.0	1.9	0.0	0.0	2.4
B3: Incorrect loop and conditional statements	7.4	0.0	1.8	5.2	15.4	8.3	0.0	2.3	1.8	4.6	1.7	1.8	3.8	5.9	0.0	0.0	2.2	3.8	0.0	4.6	2.6
B4: Inclusion of logic not in the source	3.7	13.9	5.3	5.8	0.0	12.5	1.4	4.1	1.8	3.1	0.0	0.9	3.8	0.0	3.6	4.9	4.4	1.9	5.0	2.3	3.4
B5: Removal of logic from source	3.7	5.6	8.8	2.3	0.0	0.0	2.7	1.7	0.0	3.1	1.2	0.9	3.8	11.8	3.6	3.4	8.9	13.2	2.5	2.9	3.3
B6: Mismatch of behavior after replacing API call	3.7	0.0	9.7	5.2	0.0	4.2	9.6	2.3	0.0	1.5	2.9	5.0	0.0	0.0	21.4	3.9	0.0	1.9	2.5	0.6	3.8
B: Dependency & logic bugs in the translated code	29.6	41.7	31.9	29.5	15.4	29.2	23.3	48.3	3.6	16.9	7.0	11.8	26.9	32.4	28.6	34.6	15.6	22.6	10.0	19.0	24.2
C1: Incorrect data type	7.4	27.8	8.8	21.4	7.7	4.2	8.2	15.7	1.8	15.4	7.6	8.2	0.0	26.5	7.1	21.5	11.1	1.9	7.5	0.6	11.5
C2: Incorrect input parsing	7.4	5.6	9.7	11.0	23.1	0.0	19.2	15.7	7.3	4.6	11.6	24.1	11.5	0.0	14.3	10.2	40.0	60.4	60.0	32.2	18.1
C3: Output formatting	0.0	0.0	1.8	10.4	23.1	0.0	9.6	1.7	0.0	0.0	3.5	5.0	0.0	0.0	7.1	2.4	4.4	1.9	0.0	5.2	3.9
C: Data-related Bugs	14.8	33.3	20.4	42.8	53.8	4.2	37.0	33.1	9.1	20.0	22.7	37.3	11.5	26.5	28.6	34.1	55.6	64.2	67.5	37.9	33.5
D: Model specific constraints	11.1	0.0	5.3	2.9	0.0	0.0	0.0	2.3	0.0	1.5	8.1	4.5	0.0	0.0	0.0	2.4	2.2	0.0	0.0	10.3	3.8
E: Others	0.0	2.8	14.2	4.6	0.0	25.0	19.2	6.4	0.0	0.0	0.6	1.4	0.0	0.0	7.1	5.9	2.2	0.0	7.5	6.9	5.1

latter uses 32-bit precision. The Java code thus cannot handle the large test data value, which works fine in Python.

```
------ Source Code ------
def mean_a_d(numbers: List[
        float]) -> float:
self.assertEqual(0.0,mean_a_d
        ([1e+308]))
...
```

```
 ----- Translated Code -----
public static float meanAD(List<
        Float> n) ...
void testCode() {assertEquals
        (0.0, meanAD(Arrays.asList
        (1e+308f)));}
```

Incorrect data type bugs can also occur due to differences between the type systems of the source and target PLs. For instance, when code in a dynamically typed PL such as Python is translated to a statically typed PL such as Java, preserving the behavior of a source type can be challenging.

Finding 2: A significant proportion, 30.5%, of the translation bugs occur due to syntactic and semantic differences between the source and target PLs; almost 80% of these (24.3% of all bugs) are caused by violation of target language requirements.

Almost one-third of the bugs occur due reasons, such as violating target language requirements, duplicating source syntax to the target PL, behavioral differences of APIs and operators, etc. §4.1.2 illustrates several bugs in this group. For instance, the following translated code violates syntactic constraints of the target language: the variable named `ll` in the Python code (left) is translated verbatim to C++, which results in a compilation error as `ll` is a reserved keyword in C++ (used for long-long data).

```
------ Source Code ------
ll = - 10 ** 18 - 1
```

```
 ----- Translated Code -----
ll ll = -1e18 - 1;
```

Another example of such a bug is the declaration order of methods, where some languages (e.g., Go) permit methods to call another declared subsequently, whereas others (e.g., C++) restrict calls to previously declared methods only. Thus, maintaining the wrong declaration order of methods could result in compilation errors.

Finding 2: Replacing an API call with another API call in the target PL can result in bugs.

As illustrated for the A3 bug category in §4.1.2, LLMs can incorrectly map source APIs to the APIs available in the target language. Table 4 shows that 3.3% of all the bugs fall in this category. The following example illustrates an API-mismatch bug, where the LLM replaces the Python `accumulate()` API with `IntStream.concat()` in Java, which can be used in an equivalent manner. However, the LLM erroneously adds `reduce(count).getAsInt()` to reduce the result to an integer value instead of converting the return value of `IntStream.concat()`, an integer stream, to a list/array of integers. This results in an incorrect return value in the two programs: a list of integers in Python and an integer in Java.

```
list(accumulate([0] + list(range(1,n)), count)) # Python code
int[] cumsum = IntStream.concat(IntStream.of(0), IntStream.range(1,
        n)).reduce(count).getAsInt(); # Incorrect Java code
```

Finding 3: 24.2% of the translation bugs are related to incorrect code logic and missing dependencies in the target PL, with missing imports being the dominant category.

LLMs can often translate the source logic incorrectly or miss dependencies. Missing imports is the most frequently occurring bug category (35.5% of the bugs in the group and 8.6% of the bugs overall). The other five categories in this group occur in roughly comparable numbers, ranging from 2.4% to 3.8% of the overall bugs.

4.3 Translation Bugs in Real-world Projects

Many of the bugs seen in the crafted benchmarks translation also occurred while translating Commons CLI and Click. Key among these is the removal of logic in the source (e.g., source methods and field initialization not translated, missing calls to translated methods), inclusion of logic not in source (e.g., implementation added for a stubbed method), missing imports, mismatch in API behaviors after translation, mismatch of behaviors after replacing API calls and incorrect data types.

However, real-world applications have much more complex code than crafted benchmarks, making them much harder for LLMs to translate. We found nine instances (all from Click) where the translated files contained natural-language text explaining the translation of the source file is infeasible or a non-trivial task for GPT-4. Some others included partial translations of methods in the class, leaving the rest untranslated (translation bug B5). This shows that LLMs, even with longer context windows, cannot capture dependencies between the methods implementing the class logic.

Finding 4: Real-world applications pose complex challenges for code translation, such as handling method overloading, inheritance, and code annotations, not seen in crafted datasets.

We found examples of language features used in real-world applications that LLMs can struggle with translating.

```
------ Source Code ------
public Options addOption(final Option opt) { ... }
public Options addOption(final String opt, final boolean hasArg,
      final String description) { ... }
```

```
----- Translated Code -----
def add_option(self, opt):...
def add_option_arg(self, opt, has_arg, description):...
```

For instance, Commons CLI uses method overloading frequently. GPT-4 translates these to Python in different ways, some of which are correct translations, whereas others are erroneous. For an example of the latter, there are cases where GPT-4 translates overloaded Java methods by renaming them in Python to avoid overloaded method names, leaving open the work of suitably renaming all call sites to the methods to preserve the call relations.

In another instance, overloaded methods are translated to methods with the same name, which results in broken functionality because only the last method is available as per the Python semantics, with the previous method definitions overridden.

```
------ Source Code ------
public static OptionBuilder
      hasArg() { ... }
public static OptionBuilder
      hasArg(final boolean
      hasArg) { ... }
```

```
----- Translated Code -----
@staticmethod
def has_arg():...
@staticmethod
def has_arg(has_arg):
...
```

Commons CLI and Click also illustrate the challenges posed by the use of decorators (for adding new functionality to an existing object without modifying its structure) and annotations (for adding metadata to code). For example, Click uses the `@contextmanager` decorator on a method to wrap it with a resource manager. This needs to be translated appropriately in Java to ensure automatic resource release. We also observed cases of broken inheritance relations (resulting in missing behaviors/states) in translated code and incorrect translation of exception handling.

Finding 5: The effectiveness of code translation can vary considerably based on the characteristics of the source and target PLs, such as the type system, available programming APIs, metaprogramming support via decorators or annotations, etc.

There is a clear pattern of GPT-4 performing much better in translating Commons CLI to Python than Click to Java. This is evident from not only the occurrences of successful translations (three for Commons CLI vs. none for Click) and degenerate code-generation instances (nine for Click vs none for Commons CLI), but also the translation bugs observed. This could be attributed to project-specific complexity characteristics (e.g., the largest source file has 2, 436 NCLOC in Click and 358 NCLOC in Commons CLI), and it is hard to generalize from limited observations, but language features and the available API ecosystem for a PL can have a considerable impact on the success of code translations. For example, Python-to-Java translations can be more error-prone than Java-to-Python translations, in general, because going from a dynamic type system to a static one can be harder for an LLM to reason about.

Finding 6: Although the translation bug categories remain the same, occurrences of bug instances and their distribution vary between crafted benchmarks and real-world projects.

Table 5: Successful translation (in %) of non-LLM and LLM approaches. The top-2 tools are highlighted in teal and violet.

Dataset	SL	TL	CxGo	C2Rust	Java2C#	CodeGen	CodeGeeX	StarCoder	GPT-4	Llama 2	TB-A	TB-V
CodeNet	C	Rust	-	**95.0**	-	0.0	0.0	10.5	**61.0**	0.5	0.5	0.0
	C	Go	**62.2**	-	-	34.0	0.0	12.5	**72.5**	5.0	5.5	11.0
	Java	C#	-	-	0.0	0.5	1.0	**26.5**	**49.0**	1.5	1.0	4.0
AVATAR	Java	C#	-	-	0.0	0.0	0.0	**10.4**	**59.2**	2.0	0.8	0.4

* {S, T}L: {Source, Target} Language. Data are in %. TB-{A, V}: TB-{Airoboros, Vicuna}.

The bug categories in our taxonomy are broad enough to represent translation bugs in real-world projects. However, their frequencies considerably differ. The most prevalent bugs in real-world projects are model-specific constraints (29.4%), missing imports (27.4%), and the removal of logic from the source code (15.7%), whereas for crafted benchmarks, data, syntactic, and semantic differences are more common. Moreover, the nature of these bugs is also different. For instance, in most cases, LLM's context window is not large enough to fit both project files and translated code, resulting in model-specific constraint bugs. Second, LLMs lack specific application knowledge, such as the project structure, external dependencies, and declarations outside the translation scope, i.e., the translated file. The lack of a holistic view of the application can cause bugs such as missing imports and removal of source code logic. Moreover, this limitation can cause inconsistencies in the translated code, e.g., as illustrated for Finding 4, renaming overloaded methods requires all the related call sites to be updated.

5 LLM- VS. NON-LLM-BASED TRANSLATION

This section compares LLM- and non-LLM-based code translation techniques concerning their effectiveness and the differences in translation bugs. Non-LLM-based techniques are either (1) transpilers (i.e., source-to-source compilers) or cross-lingual code executors for translation (e.g., C2Rust [9], CxGo [8], Sharpen [18], and JavaToCSharp [15]), or (2) learning-based, which leverage neural machine translation techniques to convert a code from one programming language to another (e.g., mppSMT [57] and Tree2Tree [29]).

Among the mentioned tools, we used CxGo [8] and C2Rust [9] to translate C code in our CodeNet dataset to Go and Rust, and JavaToCSharp [15] to translate Java code in CodeNet and AVATAR datasets to C#. Accordingly, we used our seven subject LLMs to translate code for all three PL pairs. Table 5 illustrates the success rate of both LLM and non-LLM approaches.

Java to C#. We observed that none of the translated code using JavaToCSharp can compile and, in fact, require heavy rewriting of library APIs (e.g., System.in and java.util.Scanner). Developers of JavaToCSharp confirmed that this tool is not built to perform full code migration. On the contrary, GPT-4 and StarCoder achieve 54.1% and 18.45% success rate in translating Java code to C#.

Finding 7: For C to Go, the best-performing LLM, i.e., GPT-4, achieves 10% higher success rate than non-LLM-based approach, whereas, for C to Rust, the non-LLM-based approach translates 95% of the code (34% better than best-performing LLM).

C to Go. CxGo transforms the C code into a common abstract syntax tree (AST) that represents both C and Go constructs. It then converts the AST into Go for translation. When compared to LLMs, we found that CxGo outperforms all the open-source models with

```
fn main() {
    let mut x = String::new();
    let mut y = String::new();
    io::stdin().read_line(&mut x).expect("Failed to read line");
    io::stdin().read_line(&mut y).expect("Failed to read line");
    let x: i32 = x.trim().parse().expect("Please type a number!");
    let y: i32 = y.trim().parse().expect("Please type a number!");
    println!("{} {}", x*y, 2*x + 2*y); }
```

(a) C Code Translated to Rust by GPT-4

```
#![allow(dead_code, mutable_transmutes, unused_assignments, unused_mut)]
extern "C" {                                                               1
        fn printf(_: *const libc::c_char, _: ...) -> libc::c_int;
        fn scanf(_: *const libc::c_char, _: ...) -> libc::c_int; }       2
unsafe fn main_0() -> libc::c_int {                                        3
        let mut x: libc::c_int = 0;
        let mut y: libc::c_int = 0;                                        4
        x = 0 as libc::c_int;
        y = 0 as libc::c_int;
        scanf(b"%d\0" as *const u8 as *const libc::c_char,
              &mut x as *mut libc::c_int);                                 5
        scanf(b"%d\0" as *const u8 as *const libc::c_char,
              &mut y as *mut libc::c_int);
        printf(b"%d %d\n\0" as *const u8 as *const libc::c_char,
               x * y, 2 as libc::c_int * x + 2 as libc::c_int * y,);
        return 0 as libc::c_int; }
```

(b) C Code Transpiled with C2Rust with annotations on safety issues.

Figure 4: Comparative analysis of C-to-Rust code translation.

an accuracy of 62.25%, but falls 10% behind GPT-4. The biggest caveat is that executing code generated by CxGo requires specific libraries, i.e., `stdio` for I/O operations instead of a more generic `fmt` library. In terms of the effect of the incorrect translation, 46.7% and 53.3% are due to compilation and runtime errors. Whereas, for the best-performing LLM, i.e., GPT-4, 76.3%, 12.7%, and 9.1% errors are related to compilation, test, and runtime failures.

With a further investigation on the type of translation bugs, we found that most GPT-4 translation bugs (71.8%) are related to unused imports, variables, and other target language-specific violations, followed by incorrect input parsing (10.3%). Whereas, with CxGo, the majority of the bugs are related to input parsing (26.7%) and mismatch of behavior after replacing API call (26.7%). In most cases, the input parsing related API, i.e., `scanf`, has been replaced by an API call that uses a custom library written specifically for the tool. Other bugs include missing definitions of API functions that are called from the translated code. The nature of these bugs is significantly different as the non-LLM-based approaches tend to have more bugs related to the custom library, while LLM-based approaches have more syntax and semantics-related bugs.

Finding 8: C2Rust generates non-idiomatic and unsafe code, whereas GPT-4 tends to generate safer and idiomatic code.

C to Rust. C2Rust is a transpiler that parses C code into C AST, converts the C AST into equivalent Rust AST by considering the differences in syntax, memory management, and ownership semantics between the two languages, and converts the Rust AST into Rust code. Code translation using C2Rust achieves a significantly higher success rate (95%) when compared to GPT-4 (61%). Using GPT-4, 50% of the translation bugs were due to compilation errors, 39.3% were caused by runtime errors, and 10.7% were due to test failures. As for C2Rust, the 5% unsuccessful translations were due to compilation errors, the majority of them caused by unused imports.

A close inspection of the translated Rust code offers several noteworthy observations. Figure 4 shows a typical translation scenario. For the same C code, the translation by GPT-4 (Figure 4-(a)) adheres to idiomatic Rust translations, whereas translations produced by C2Rust (Figure 4-(b)) contain unsafe code that directs the Rust compiler to bypass any safety checks. In fact, all of the Rust2C-translated code samples are unsafe, whereas, only three programs translated by GPT-4 contain unsafe code. We list some of the issues with the unsafe Rust code.

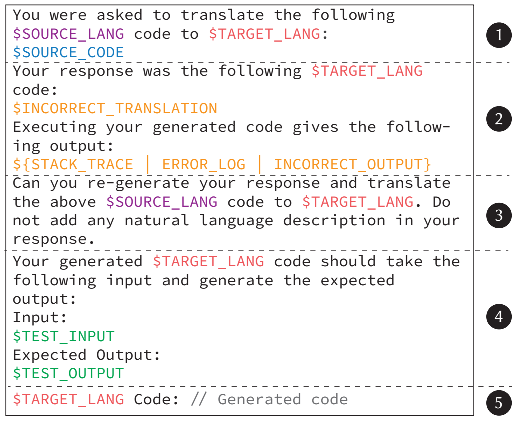
- *Safety risks from compiler directives.* The use of compiler directives (see 1 in Figure 4-(b)) can introduce various safety risks: (1) permitting dead code, (2) the `mutable_transmutes` directive can cause undefined behaviors and mask logical flaws, and (3) allowing unnecessary mutability can cause race conditions.
- *Use of* `extern "C"` *for foreign function interface.* Potential safety risks may result due to reliance on external C (see 2 in Figure 4-(b)) functions such as `printf` and `scanf` without Rust's safety guarantees. Specifically, the use of `scanf` (as in 5) without buffer size specifications can lead to vulnerabilities such as buffer overruns.
- *Unrestricted use of unsafe block.* The `unsafe` block in `main` function (see 3 in Figure 4-(b)) circumvents Rust's safety checks for the code encompassed in that method. This increases the risk of memory safety violations. This issue is further exacerbated by the use of implicit casting (e.g., `0 as libc::c_int` as in 4) in the method body, which can cause unexpected behavior.

6 MITIGATING TRANSLATION BUGS

In this section, we discuss how context information pertaining to unsuccessful translations can help fix buggy translations.

Prompt Crafting. Inspired by other works on prompt crafting for fixing bugs [76, 84] and how human developers would address a translation error, we propose an *iterative* prompting approach. Our hypothesis is that providing more context information to LLMs can help generate better code. Based on this hypothesis, we include the following contextual information in the revised prompts (Figure 5).
❶ **Source code and original prompt.** Here, we include the original code and the previous prompt used for translating the original code to remind LLMs about the previous task.
❷ **Incorrect translation and error details.** Here, we provide the incorrectly translated code (`$INCORRECT_TRANSLATION`), and details regarding the outcome of the translation. If it is runtime error, we provide stack trace (`$STACK_TRACE`); for compilation error, we provide error log (`$ERROR_LOG`); for test failures, we provide the incorrect output (`$INCORRECT_OUTPUT`); finally, for non-terminating execution, we provide a custom message "The program enters infinite loop."
❸ **Instructions for translation.** Here, we ask LLM to mitigate the bug and avoid including natural-language text in the response.
❹ **Expected behavior.** This part is used optionally if the prior translation was a functional error. Here, we provide test input and expected output pair for the previously wrong output.
❺ **Model-specific keyword.** This part is specific to code LLMs and contains the name of the target PL following code LLM templates. All the prompts that we used are in the replication package [7].

Iterative Translation Bug Mitigation. Our prompt-crafting technique is *iterative*. At each iteration $iter_i$, we update the prompt template (Figure 5) with information corresponding to the previously failed translation. We refer to the outcome of $iter_i$ translation as *translation patch*. At the end of each iteration, we verify if the patch results in a successful translation. If not, we utilize the outcome of the patch and build the prompt for the next iteration. The iterative mitigation can continue for a fixed number of iterations



```
You were asked to translate the following
$SOURCE_LANG code to $TARGET_LANG:
$SOURCE_CODE                                          1
-------------------------------------------------
Your response was the following $TARGET_LANG
code:
$INCORRECT_TRANSLATION
Executing your generated code gives the follow-       2
ing output:
${STACK_TRACE | ERROR_LOG | INCORRECT_OUTPUT}
-------------------------------------------------
Can you re-generate your response and translate
the above $SOURCE_LANG code to $TARGET_LANG. Do       3
not add any natural language description in your
response.
-------------------------------------------------
Your generated $TARGET_LANG code should take the
following input and generate the expected
output:
Input:                                                4
$TEST_INPUT
Expected Output:
$TEST_OUTPUT
-------------------------------------------------
$TARGET_LANG Code: // Generated code                  5
```

Figure 5: Prompt crafting template for LLMs with the context corresponding to the outcomes of unsuccessful translation.

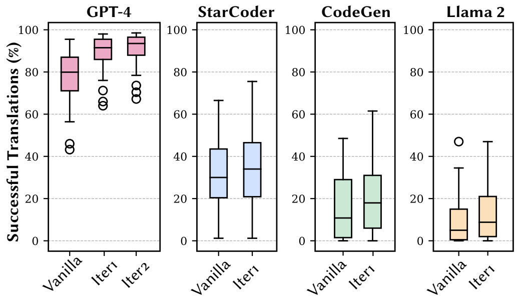


Figure 6: Effectiveness of prompt crafting in mitigating translation bugs over LLMs.

or until the percentage of successful translations at the previous iteration is smaller than a pre-defined threshold.

To evaluate the effectiveness of our iterative prompt crafting, we attempted to mitigate unsuccessful translations from RQ1 for four subject LLMs: CodeGen, StarCoder, GPT-4, and Llama 2. We excluded CodeGeeX because its prompt template is rigid, and we could not introduce additional contexts. Due to inferior performance in vanilla prompting, we also excluded the TB-Airoboros and TB-Vicuna. We set the termination criteria so that the mitigation process terminates if the overall increase of successful translation is less than 5%. Figure 6 summarizes our findings. Based on the proposed termination criterion, our mitigation process lasted for two iterations for GPT-4, and one iteration for the rest of the models. The results suggest that the proposed technique can increase the number of successful translations for all the studied LLMs, with $iter_1$—prompting on results from vanilla prompts—increasing the success rate of GPT-4, StarCoder, Codegen, and Llama 2 by 12.33%, 3.55%, 2.65%, 1.97%, respectively. $iter_2$—prompting on results from $iter_1$, can improve GPT-4 by 1.7%.

We also wanted to understand how translation bugs evolve during this iterative prompting process. To that end, we tracked the error outcomes of unsuccessful translations (§3.2) from vanilla prompting to $iter_1$ (for all models) and from $iter_1$ to $iter_2$ (for GPT-4). Figure 7 illustrates the results of our deeper analysis of GPT-4 and the results for the other models are available in our artifact [7].

With $iter_1$, we observe a substantial reduction in *compilation errors* for GPT-4, with 58% completely fixed (no error) and 38% transformed to other translation bugs. However, for other errors, the

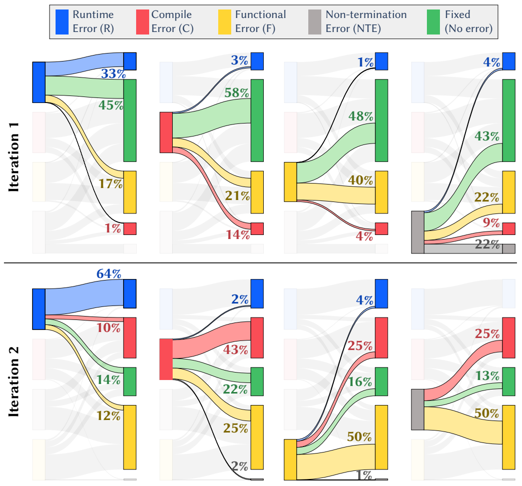


Figure 7: Translation outcomes after prompting GPT-4.

Table 6: Characteristics of LLM-based and non-LLM-based code translation approaches.

Characteristics	Non-LLM	LLM
Broader context	✓	X
Determinism and reasoning	✓	X
Leveraging target-language idioms	X	✓

percentage is lower—45% runtime errors, 48% functional errors, and 43% non-terminating executions—suggesting these bugs are harder to mitigate. With $iter_2$, the outcome of most bugs does not evolve and they remain the same. In both cases, we observe a few cases where the outcome of the translation degrades: i.e., functional error transforms to runtime/compilation error, or runtime error transforms to compilation error. Also, we found several instances where, instead of fixing the bug, the LLM introduced a new one. These findings show that, although the proposed technique improves overall effectiveness, future work should be directed toward combining the power of program analysis and more nuanced LLM approaches, e.g., by including more suitable code examples for in-context learning.

7 DISCUSSION

Pros and Cons of LLM- and non-LLM-based Translation. We discuss strengths and weaknesses of each approach with respect to three key characteristics (Table 6). First, non-LLM-based approaches, specifically transpilers, have a more comprehensive context of an application. On the other hand, while translating a module, the variable types, method signatures, folder structure, and dependencies associated with that module are often unavailable to LLMs because of their restriction on the token size. Second, non-LLM-based techniques are generally deterministic, allowing for more predictable reasoning, whereas LLMs are inherently probabilistic, with a greater degree of creativity. The third characteristic pertains to the naturalness of the translated code. In this respect, non-LLM-based approaches mostly do not leverage target language idioms, which can negatively affect code readability and maintainability. In some cases, this even leads to security issues (inC case of C to Rust). Conversely, LLMs tend to generate more natural, human-like code. More research on utilizing the two approaches to leverage their strengths would be fruitful.

Translating real-world projects. Unlike crafted benchmarks, files in real-world projects do not exist as standalone programs, and providing relevant context about inter-file dependencies can help an LLM produce better code. More fundamentally, however, new techniques are required to enable code translation to scale to real-world applications and also generate high-quality translations. For instance, such techniques could use program analysis to provide more context while translating a piece of code. Another potential direction is to leverage program-decomposition techniques to split the source file into smaller fragments, each translated separately via prompts that encode appropriate context information about the fragment's dependencies; the translated fragments are then composed to produce the fully translated code. Finally, a significant challenge in translating real-world projects is handling library API calls and the differences in the API ecosystem of different PLs. There can be cases where no suitable API exists in the target PL to translate an API call to—techniques that combine code summarization and code synthesis could be investigated to fill such gaps.

Improving open and closed-source LLMs. The findings of this work show considerable scope for improving open-source and close-source LLMs for code translation. For closed-source models (i.e., GPT-4), the increasing complexity of use cases (i.e., translating real-world projects) calls for an evolution in prompting strategies perhaps with the help of program analysis. One promising research direction could involve creating prompts that build upon each other [78, 87], following a coherent and logical flow of information. For open-source LLMs, enhancing the performance could involve fine-tuning [39, 71], wherein different models [42] (instead of one model fits all) are carefully trained to tackle distinct aspects of the translation process (particularly those related to the bug categories).

8 THREATS TO VALIDITY

Like any empirical study, there are threats to the validity of our results. We discuss the most significant ones of these, along with the mitigating factors.

External Validity. Threats to external validity pertain to whether our results can generalize to other experimental settings. Key factors here include the PLs, LLMs, and datasets selected. To mitigate these threats, we selected five PLs, guided by PL popularity [21] while also covering different programming paradigms. In terms of datasets, we used multiple well-known datasets with different characteristics and also two real-world projects. For LLM selection, we included several state-of-the-art general and code LLMs. Finally, for non-LLM-based code translation, we used multiple tools covering different PLs and translation approaches.

Internal Validity. As for threats to internal validity, one potential threat is that, for each translation task, we performed the translation once. Performing a translation task multiple times may change the success rates in translation as LLMs are inherently non-deterministic. However, this does not affect the primary goal of our work, which is to identify the characteristics of translation bugs and provide insights to help drive future research. Regardless of the number of repetitions and randomness, the nature of an unsuccessful translation remains unchanged. Moreover, to reduce the effects of LLMs' sensitivity to prompt templates, we followed the best practices described in the respective artifacts/papers/reports (refer §2). Another threat to internal validity is that we do not have a formal inter-rater reliability metric for our manual labeling of translation bugs. To mitigate this threat, after each labeling round, both labelers met and discussed any discrepancies and resolved them through discussion. At the end, each bug was assigned a single mutually agreed label. On top of that, one author went through the entire labeling to ensure consistency among the groups. Finally, our results may be affected by bugs in our automation scripts. To address this threat, we thoroughly tested the scripts and spot-checked the results for correctness. Moreover, we make our artifacts publicly available [7] to enable review and replication of our results.

Construct Validity. We measured translation correctness using the test cases provided for the programs in our datasets, which is a commonly used approach for assessing the quality of generated code. This approach inherently comes with the risk that inadequate or weak test suites can cause buggy translations that pass the test suites to be considered correct. We note that one of our datasets, CodeNet [63], contains one test case per sample and may be susceptible to this threat. However, the other datasets used contain fairly rigorous test suites, which mitigates this threat to validity.

9 RELATED WORK

Code Translation and Synthesis. There exist two broad classes of code-translation approaches. The first category includes tools such as transpilers, or source-to-source compilers, that leverage program-analysis techniques for converting code from one PL to another. For instance, C2Rust [9] and CxGo [8] are transpilers that translate C programs to Rust and Go, respectively, whereas Sharpen [18] and Java2CSharp [14] convert Java code to C#. The second category includes learning-based techniques, including lexical statistical machine translation [55, 56] and tree-based neural networks [29] for translating Java to C#. Other work in the category leverages deep learning and unsupervised learning [50, 65] for translating C++, Java, and Python code, phase-based statistical learning [47] for C#-to-Java translation, etc. More recent techniques leverage LLMs (e.g., StarCoder [52], PolyCoder [86], SantaCoder [24], CodeGen [58], BLOOM [67], CodeT5 [77], CodeX [27], GPT-4 [12], Llama 2 [16], etc.) for code generation and code translation (CodeGeeX [91]). However, in the context of LLMs, there is no study that understands the bugs introduced by LLMs during code-translation tasks.

Bug and repair study. Software bugs are well studied [25] including several works on deep learning model-related bugs [40, 41, 90]. Moreover, there is also an extensive list of works on fixing software bugs using LLMs (e.g., [32, 45, 84, 85], and other approaches (e.g., [33, 83, 88]). Compared to both of these classes of studies, here, we study bugs introduced by LLMs while translating code from one PL to another. Also, we compare the effectiveness of LLM with non-LLM-based approaches in terms of code translation task.

10 CONCLUDING REMARKS

Code translation has various applications, from modernizing enterprise applications to migrating legacy software to modern PLs. Given the promising performance of LLMs in code synthesis, we were interested to understand how they perform in code translation task. Our empirical investigation of the general and code LLMs across five PLs and several benchmarks and real-world projects demonstrates that state-of-the-art LLMs are yet to effectively automate code translation, specifically to translate complex real-world projects. Through meticulous manual analysis, we also identified 15 root causes that make LLMs produce unsuccessful translations. Furthermore, to investigate how providing more context can help LLMs generate better code, we presented and evaluated an iterative prompt-crafting technique. We also assessed existing non-LLM-based techniques, comparing their strengths and weaknesses. We are currently considering several directions for future work. First, we want to investigate how existing SE/PL techniques can help mitigate these bugs, including giving more context to LLMs. Second, our ultimate goal is to advance real-world project translation.

REFERENCES

[1] 2018. Upgrading GitHub from Rails 3.2 to 5.2. https://github.blog/2018-09-28-upgrading-github-from-rails-3-2-to-5-2/.

[2] 2020. Supporting Linux kernel development in Rust. https://lwn.net/Articles/829858/.

[3] 2020. Transform monolithic Java applications into microservices with the power of AI. https://developer.ibm.com/tutorials/transform-monolithic-java-applications-into-microservices-with-the-power-of-ai/.

[4] 2020. Will code move on to a language such as Rust? https://www.theregister.com/2020/06/30/hard_to_find_linux_maintainers_says_torvalds/.

[5] 2021. GitHub's Journey from Monolith to Microservices. https://www.infoq.com/articles/github-monolith-microservices/.

[6] 2023. Apache Commons CLI. https://commons.apache.org/proper/commons-cli/.

[7] 2023. Artifact Website. https://github.com/Intelligent-CAT-Lab/PLTranslationEmpirical.

[8] 2023. C to Go Translator. https://github.com/gotranspile/cxgo.

[9] 2023. C2Rust Transpiler. https://github.com/immunant/c2rust.

[10] 2023. Click. https://click.palletsprojects.com/en/8.1.x/.

[11] 2023. CodeGeeX. https://github.com/THUDM/CodeGeeX/blob/main/tests/test_prompt.txt.

[12] 2023. GPT-4 Technical Report. https://cdn.openai.com/papers/gpt-4.pdf.

[13] 2023. Hugging Face Open LLM Leaderboard. https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard.

[14] 2023. Java 2 CSharp Translator for Eclipse. https://sourceforge.net/projects/j2cstranslator/.

[15] 2023. Java to CSharp Converter. https://github.com/paulirwin/JavaToCSharp.

[16] 2023. Llama-2. https://ai.meta.com/research/publications/llama-2-open-foundation-and-fine-tuned-chat-models/.

[17] 2023. py_compile—Compile Python source files. https://docs.python.org/3/library/py_compile.html.

[18] 2023. Sharpen - Automated Java->C# coversion. https://github.com/mono/sharpen.

[19] 2023. TheBloke Airoboros 13B. https://huggingface.co/TheBloke/airoboros-13B-HF.

[20] 2023. TheBloke Wizard Vicuna 13B. https://huggingface.co/TheBloke/Wizard-Vicuna-13B-Uncensored-HF.

[21] 2023. TIOBE Index. https://www.tiobe.com/tiobe-index/.

[22] Seif Abukhalaf, Mohammad Hamdaqa, and Foutse Khomh. 2023. On Codex Prompt Engineering for OCL Generation: An Empirical Study. *arXiv preprint arXiv:2303.16244* (2023).

[23] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2021. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590* (2021).

[24] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988* (2023).

[25] Boris Beizer. 1990. Software testing techniques.

[26] Alexander Bergmayr, Hugo Bruneliere, Javier Luis Cánovas Izquierdo, Jesús Gorronogoitia, George Kousiouris, Dimosthenis Kyriazis, Philip Langer, Andreas Menychtas, Leire Orue-Echevarria, Clara Pezuela, et al. 2013. Migrating legacy software to the cloud with ARTIST. In *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 465–468.

[27] 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).

[28] Pinzhen Chen and Gerasimos Lampouras. 2023. Exploring data augmentation for code generation tasks. *arXiv preprint arXiv:2302.03499* (2023).

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

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

[31] Aryaz Eghbali and Michael Pradel. 2022. CrystalBLEU: precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[32] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[33] L. Gazzola, D. Micucci, and L. Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 01 (jan 2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[34] 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.

[35] Linyuan Gong, Jiayi Wang, and Alvin Cheung. 2023. ADELT: Transpilation Between Deep Learning Frameworks. *arXiv preprint arXiv:2303.03593* (2023).

[36] 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.

[37] DongNyeong Heo and Heeyoul Choi. 2022. End-to-End Training of Both Translation Models in the Back-Translation Framework. *arXiv preprint arXiv:2202.08465* (2022).

[38] Jaemin Hong. 2023. Improving Automatic C-to-Rust Translation with Static Analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 273–277.

[39] Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146* (2018).

[40] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 510–520.

[41] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing deep neural networks: Fix patterns and challenges. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1135–1146.

[42] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. 1991. Adaptive mixtures of local experts. *Neural computation* 3, 1 (1991), 79–87.

[43] Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. 2023. Attention, Compilation, and Solver-based Symbolic Analysis are All You Need. *arXiv preprint arXiv:2306.06755* (2023).

[44] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689* (2023).

[45] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.

[46] Anup K Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. 2021. Mono2micro: a practical and effective tool for decomposing monolithic java applications to microservices. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 1214–1224.

[47] Svetoslav Karaivanov, Veselin Raychev, and Martin 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*. 173–184.

[48] Justas Kazanavičius and Dalius Mažeika. 2019. Migrating legacy software to microservices architecture. In *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*. IEEE, 1–5.

[49] 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.

[50] Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A deobfuscation pre-training objective for programming languages. *Advances in Neural Information Processing Systems* 34 (2021), 14967–14979.

[51] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[52] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[53] Fang Liu, Jia Li, and Li Zhang. 2023. Syntax and Domain Aware Model for Unsupervised Program Translation. *arXiv preprint arXiv:2302.03908* (2023).

[54] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language

models for code generation. *arXiv preprint arXiv:2305.01210* (2023).

[55] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 651–654.

[56] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2014. Migrating code with statistical machine translation. In *Companion Proceedings of the 36th International Conference on Software Engineering*. 544–547.

[57] 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 *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 585–596.

[58] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*.

[59] Vikram Nitin, Shubhi Asthana, Baishakhi Ray, and Rahul Krishna. 2022. CARGO: ai-guided dependency analysis for migrating monolithic applications to microservices architecture. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[60] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[61] Hongyu Pei Breivold. 2020. Towards factories of the future: migration of industrial legacy automation systems in the cloud computing and Internet-of-things context. *Enterprise Information Systems* 14, 4 (2020), 542–562.

[62] Ricardo Pérez-Castillo, Manuel A Serrano, and Mario Piattini. 2021. Software modernization to embrace quantum technology. *Advances in Engineering Software* 151 (2021), 102933.

[63] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

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

[65] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems* 33 (2020), 20601–20611.

[66] Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773* (2021).

[67] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100* (2022).

[68] Rajaraajeswari Settu and Pethuru Raj. 2013. Cloud application modernization and migration methodology. *Cloud Computing: Methods and Practical Approaches* (2013), 243–271.

[69] André Silva, João F Ferreira, He Ye, and Martin Monperrus. 2023. MUFIN: Improving Neural Repair Models with Back-Translation. *arXiv preprint arXiv:2304.02301* (2023).

[70] Ishan Mani Subedi, Maninder Singh, Vijayalakshmi Ramasamy, and Gursimran Singh Walia. 2021. Application of back-translation: a transfer learning approach to identify ambiguous software requirements. In *Proceedings of the 2021 ACM Southeast Conference*. 130–137.

[71] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. 2019. How to fine-tune bert for text classification?. In *Chinese Computational Linguistics: 18th China National Conference, CCL 2019, Kunming, China, October 18–20, 2019, Proceedings 18*. Springer, 194–206.

[72] Qiushi Sun, Nuo Chen, Jianing Wang, Xiang Li, and Ming Gao. 2023. TransCoder: Towards Unified Transferable Code Representation Learning Inspired by Human Skills. *arXiv preprint arXiv:2306.07285* (2023).

[73] Marc Szafraniec, Baptiste Roziere, Hugh Leather Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code translation with compiler representations. *arXiv preprint arXiv:2207.03578* (2022).

[74] Johannes Thönes. 2015. Microservices. *IEEE software* 32, 1 (2015), 116–116.

[75] Chih-Kai Ting, Karl Munson, Serenity Wade, Anish Savla, Kiran Kate, and Kavitha Srinivas. 2023. CodeStylist: A System for Performing Code Style Transfer Using Neural Networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 16485–16487.

[76] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[77] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[78] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[79] Justin D Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. 2021. Perfection not required? Human-AI partnerships in code translation. In *26th International Conference on Intelligent User Interfaces*. 402–412.

[80] Justin D Weisz, Michael Muller, Steven I Ross, Fernando Martinez, Stephanie Houde, Mayank Agarwal, Kartik Talamadupula, and John T Richards. 2022. Better together? an evaluation of ai-supported code translation. In *27th International Conference on Intelligent User Interfaces*. 369–391.

[81] Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xing Hu, Zidong Du, Ling Li, et al. 2022. BabelTower: Learning to Auto-parallelized Program Translation. In *International Conference on Machine Learning*. PMLR, 23685–23700.

[82] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[83] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2022. Practical program repair in the era of large pre-trained language models. *arXiv preprint arXiv:2210.14179* (2022).

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

[85] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246* (2023).

[86] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

[87] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601* (2023).

[88] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-based Automated Program Repair. *arXiv preprint arXiv:2301.03270* (2023).

[89] WG Zhang, Arne J Berre, Dumitru Roman, and Hans Aage Huru. 2009. Migrating legacy applications to the service Cloud. In *Proceedings of the 14th Conference Companion on Object Oriented Programming Systems Languages and Applications*. 59–68.

[90] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 129–140.

[91] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568* (2023).

[92] Terry Yue Zhuo, Zhuang Li, Yujin Huang, Yuan-Fang Li, Weiqing Wang, Gholamreza Haffari, and Fatemeh Shiri. 2023. On robustness of prompt-based semantic parsing with large pre-trained language model: An empirical study on codex. *arXiv preprint arXiv:2301.12868* (2023).