

Evaluating Quality of Code Translation

Thesis submitted by

Shashank G (2022AIB2684)

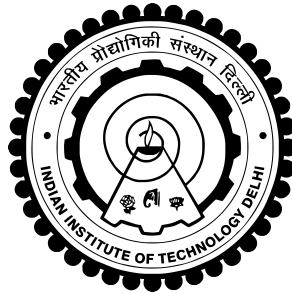
under the guidance of

**Prof. Srikanta Bedathur Jagannath (Indian Institute of Technology
Delhi)**

**Dr. Yogish Sabharwal and Dr. Anamitra Roy Choudhury (IBM
India Research Lab)**

*in partial fulfilment of the requirements
for the award of the degree of*

Bachelor and Master of Technology



Department Of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY DELHI

January 2024

THESIS CERTIFICATE

This is to certify that the thesis titled **Evaluating Quality of Code Translation**, submitted by **Shashank G (2022AIB2684)**, to the Indian Institute of Technology, Delhi, for the award of the degree of **Masters of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Srikanta Bedathur
Dept. of Computer Science
IIT-Delhi, 600 036

Place: New Delhi

Date: **5th June 2024**

ACKNOWLEDGEMENTS

We would like to express our deepest appreciation to our advisor, **Prof. Srikanta Bedathur**, whose expertise, understanding, and patience have significantly enriched our graduate experience. Our heartfelt thanks also go to Dr. Yogish Sabharwal and Dr. Anamitra Roy Choudhury from IBM India Research Lab, and Monika, for their valuable insights and feedback. The regular weekly meetings with the IBM Research team were incredibly educational and beneficial. We are also grateful for the computing resources provided for this project.

Lalit and Shashank have contributed equally to this project, and we are both grateful to each other for our contributions and valuable work on this project.

DECLARATION

This Master’s project was completed by a team of two members under the supervision of Prof. Srikanta Bedathur at IIT Delhi. My work was concentrated on program analysis-based approaches and methods to improve the performance of prompt-based techniques. Additionally, I emphasized quantitatively evaluating translation quality and developing the evaluation metric for it. I collaborated with Lalit Meena, and we contributed equally to the project, appreciating each other’s efforts and valuable work. His primary focus was on prompt-based approaches and software engineering practices to enhance functional correctness while preserving the structural and logical integrity of the translated program. I have also frequently cited Lalit’s work as [Mee24].

ABSTRACT

Code translation has long posed a challenge for the AI community but is essential for organizations and individual developers seeking to migrate legacy codebases to modern languages or translate important libraries present in one language to another. The scarcity of real-world parallel corpora containing source and target translation pairs exacerbates the problem. Since their advent, LLMs have been successful in various tasks such as code generation, summarization, and question-answering etc. Chain-of-Thought (CoT) reasoning, a technique that enables LLMs to systematically reason through tasks to arrive at a solution, has demonstrated remarkable success without requiring fine-tuning. However, its application in translating source code from one language to another remains unexplored. In this work, we investigate approaches utilizing CoT and propose a metric to evaluate the structural resemblance between source and target code. Our method, CodeTransCoT, significantly outperforms the Vanilla method, with performance improvements from 18% to 42% in Java-Python translation on the Avatar dataset and from 9% to 53% in Python-Java translation on the CodeNet dataset. Additionally, our approach maintains the structural quality and logic of the source program, as demonstrated by the proposed metric, thereby reducing compilation errors.

Contents

ACKNOWLEDGEMENTS	i
DECLARATION	ii
ABSTRACT	iii
LIST OF TABLES	vi
LIST OF FIGURES	vii
ABBREVIATIONS	viii
NOTATION	ix
1 INTRODUCTION	1
2 Problem Statement	4
3 Related Work	5
4 Challenges in Code Translation	7
4.1 Bug Categories: Java \rightleftharpoons Python Translations	9
4.1.1 Compilation or Runtime Failed Bugs	9
4.1.2 Test Case Failed Bugs	11
4.2 Translation Quality Bugs	12
5 METHODOLOGY	17
5.1 Vanilla Approach	17
5.1.1 Discussions	18
5.2 Additional Context based Approach	18
5.2.1 Type information based Approach	18
5.2.2 Dynamic Function Trace	19

5.3	Program analysis based approach	20
5.3.1	More Details	21
5.4	Iterative Improvement	23
6	Evaluation Metrics	25
6.1	Pass@ k estimator	25
6.2	Quality Metric	26
6.2.1	Call Graph	26
6.2.2	Graph Edit Distance	27
7	Experiments	30
7.1	Models	30
7.2	Datasets	30
7.3	Results	31
7.3.1	Evaluations of the Code Translation Quality	32
8	Conclusion	35
8.1	Conclusion	35

List of Tables

7.1	Results of the various approaches mentioned in Methodology 5	31
7.2	Iterative technique results	32
7.3	Quality metric results for the Vanilla and CodeTransCoT approach for all the models. The reported number is average graph edit distance across the whole dataset	32

List of Figures

4.1	Categorisation of Translation Bugs Introduced by LLMs.	8
4.2	Comparison of percentage of bugs of each category in translated Python code. Benchmark: AVATAR.	10
4.3	Comparison of percentage of bugs of each category in translated Java code. Benchmark: AVATAR.	10
4.4	Example of an input Java source code.	13
4.5	Python translation of Java code in figure 4.4, as generated by GPT-4 under vanilla prompting technique.	14
4.6	Example of an input Python source code.	15
4.7	Java translation of Python code in figure 4.6, as generated by StarCoder under vanilla prompting technique.	15
5.1	Vanilla prompting template.	17
5.2	Additional Context-based Approach	18
5.3	Type information based prompt	19
5.4	Dynamic Function Trace Prompt	20
5.5	Workflow of the Iterative prompt approach	24
6.1	Comparison of structural representations between a Java source code and its Python translation using the Vanilla approach through the StarCoder LLM.	26
6.2	Corresponding Java call graph (left) and Python call graph (right) of the Code in 6.1	27
6.3	Correcting the call graph	28
6.4	Reference program for the source Java program in Figure 6.2	28
6.5	Reference Java program for quality analysis of Python code in figure 4.6	29
7.1	Number of nodes vs Average graph edit distance plots. This is on Avatar dataset and translation was from Python to Java	33
7.2	Number of nodes vs Average graph edit distance plots. This is on Avatar dataset and translation was from Java to Python	34

ABBREVIATIONS

IITD	Indian Institute of Technology, Delhi
IBM	International Business Machines
RTFM	Read the Fine Manual
LLM	Large Language Model
LLMs	Large Language Models
DNN	Deep Neural Network

NOTATION

<< ... >> - This notation can be used like this << *content_to_fill* >>. It means if it is used inside the input prompt template then it should be appropriately filled to complete the prompt else the content inside will be generated by LLM model according to output template.

Chapter 1

INTRODUCTION

Accurate and effective translation of source code from one programming language to another has become an essential challenge within AI research communities. Organizations and developers also frequently encounter the need to translate code in their work. This necessity arises from various factors, such as the need to migrate legacy codebases to modern languages to enhance performance, maintainability, or compatibility with current technologies. Code translation is also helpful when you need to make libraries which is present in another language. Additionally, translating code from strongly typed languages like Java to dynamically typed languages like Python, which offer low maintenance costs and ease of understanding, has become a common requirement. This translation process can be labor-intensive and error-prone when done manually, underscoring the need for automated solutions.

Large language models (LLMs) have significantly enhanced tasks such as natural language understanding, text generation, and machine translation, improving accuracy and coherence in these areas. They also present a promising approach to automating code translation, offering a potentially less expensive and more scalable solution. However, there are many challenges associated in code translation with the help of LLMs. Firstly, there is a scarcity of real-world parallel corpora containing source and target translation pairs, which are essential for fine-tuning LLMs. Even when such data is available, fine-tuning large models with billions of parameters demands substantial computational resources that are not readily accessible to everyone. Despite their potential, most LLMs still struggle to generate consistently accurate and reliable translated code.

The primary focus of past research in this area has been on improving the execution accuracy of translated code, ensuring that the translated code performs the same functions as the original. Lachaux et al. [LRCL20] address the challenge of limited parallel corpora in this field. They utilize techniques such as Cross-lingual Masked Language Model pretraining, Denoising auto-encoding, and Back-translation to develop a fully unsupervised neural transcompiler, demonstrating high accuracy in translating functions between C++, Java, and Python. Similarly, Szafraniec et al. [SRL⁺23] utilize these techniques but further enhance the dataset with LLVM IR (Intermediate Representations) to better capture the similarities between programs with different semantics in various languages. Xie et al. [XNFR23] explore data augmentation techniques by constructing comparable corpora (parallel corpora where the source-target pairs have similar functionality) and augmenting available parallel data with different reference translations to increase target translation

variability. Other researchers have also employed reinforcement learning to address this problem. Jana et al. [JJJ⁺24] integrate reinforcement learning into the fine-tuning process, incorporating compiler feedback and symbolic execution (symexec)-based equivalence testing to ensure functional equivalence between input and output programs. Additionally, PPOCoder [PSR23] is an RL-based code translation framework that leverages CodeBLEU-inspired feedback in the LLM training process.

All previous approaches involve fine-tuning or reinforcement learning feedback, which, while effective, are not user-friendly for those inexperienced with fine-tuning LLMs. Additionally, no prior work has analyzed the types of errors occurring in test sets. Pan et al. [PIK⁺24] conducted a large-scale evaluation and taxonomy of translation bugs, identifying that many bugs stem due to the difference in syntax and semantic difference between two languages, missing dependencies, incorrect logic implementation, and mishandling of input-output. We believe that effective mitigation strategies can only be developed by first classifying the different types of bugs encountered and by distinguishing programming languages based on their paradigms (e.g., strongly typed, loosely typed, object-oriented).

Considering the unresolved challenges, we have selected Java and Python for translation, as Java is object-oriented strongly typed and Python is multi-paradigm dynamically typed. Previous research [PIK⁺24] has shown that Java-Python and Python-Java translations have historically underperformed, making them ideal candidates for further research. Additionally, Java and Python are among the most widely used programming languages globally, increasing the relevance and impact of our research. We use the CodeNet and Avatar datasets, which contain code written by competitive programmers and submitted to various online judges, including AtCoder, AIZU, Google Code Jam, Codeforces, GeeksforGeeks, and LeetCode. Our approaches relies solely on prompt-based methods, eliminating the need for any fine-tuning. We evaluate the translated programs based on functional correctness.

While execution accuracy is a crucial parameter for judging code translation, it is not the only important factor, as it is essential that the translated code maintains structural resemblance to the original to ensure a smooth transition from one language to another. Having structural similarity also helps preserve function-wise logic of the source code leading to correct translations. To measure this structural similarity score, we propose a quality metric to measure the quality of the translation.

We present various approaches, such as explanation-based and pseudocode methods, where we include explanations or pseudocode of the source program as additional information in the prompt. Additionally, we implement In-Context Learning by selecting samples manually, which significantly reduces compilation errors. Our efforts then shift to Comp-WiseTrans, a Chain-of-Thought based approach, aiming to preserve the structural similarity to the source program. Ultimately, we achieve impressive results with the CodeTransCoT technique. We found that reducing compilation errors significantly boosts performance, and

maintaining structural resemblance which aids in generating compilation-error-free code.

Our main contributions are as follows:

- We introduce CodeTransCoT, a novel Chain-of-Thought based technique designed to efficiently and accurately translate code from one language to another while preserving structural similarity.
- We conduct an extensive study of this approach using various models such as StarCoder, CodeLlama, Granite-8B, and Granite-20B, utilizing the Avatar and CodeNet datasets for their accessibility and the availability of test cases for evaluating functional correctness. We report Pass@1*, Pass@1, and Pass@5 metrics for all settings and analyze the types of errors generated.
- We propose a Quality metric to measure structural similarity and verify if the generated program matches the intended output. This involves performing static analysis of the code and extracting call graph information.

Chapter 2

Problem Statement

Multilingual Code Translation is the process of converting source code written in one programming language into an equivalent code in another language while preserving its functionality and readability. Traditional methods for code translation involve manual rewriting or rule-based systems, both of which are time-consuming and error-prone.

This work investigates the use of LLMs to improve the quality of code translation i.e. to improve the ability to accurately convey the intended functionality of the input code. As errors in translation can result in incorrect behavior, inefficiencies, or non-functional code. This work focuses on improving the quality and accuracy of code translation using state-of-the-art Large Language Models (LLMs).

Chapter 3

Related Work

Rule-based transpilers, or handcrafted rule-based transpilers, typically utilize traditional compiler techniques and concepts such as parsing and abstract syntax trees. Examples include `java2python` [TMS16] and `py2java` [Fom19]. The TSS code converter [tss23] is a commercial J2P transpiler. However, many of these tools include disclaimers noting that the translated code may not compile or run without further adjustments.

Lachaux et al. [LRCL20] tackle the challenge of limited parallel corpora by leveraging advanced techniques in unsupervised machine translation. They employ Cross-lingual Masked Language Model pretraining, Denoising auto-encoding, and Back-translation to develop a fully unsupervised neural transcompiler. Their model demonstrates high accuracy in translating functions across C++, Java, and Python. They introduce a test set comprising 852 parallel functions and unit tests to verify translation correctness, showing superior performance over rule-based commercial baselines. Building on this foundation, Szafraniec et al. [SRL⁺23] expand the dataset with LLVM IR (Intermediate Representations) to enhance capturing semantic similarities between programs across different languages. They augment existing test sets for code translation by including hundreds of functions from Go and Rust.

Xie et. al. [XNFR23] explore data augmentation techniques by constructing comparable corpora (parallel corpora where the source-target pairs have similar functionality) and augmenting available parallel data with different reference translations. They build and analyze multiple types of comparable corpora, such as Naturally available corpora, Generated comparable corpora and even Random Comparable Corpora. Furthermore, they automatically generate additional translation references for available parallel data to reduce overfitting to a single reference translation and increase variation in target translations.

Previous studies have also explored the application of reinforcement learning to address this challenge. Jana et. al. [JJJ⁺24] train an LLM via reinforcement learning, by modifying the fine-tuning process to incorporate compiler feedback and symbolic execution (symexec)-based equivalence testing feedback that checks for functional equivalence between the input and output program. They achieve the best-known results on the Avatar-TC dataset, with 53% accuracy for Java-Python translations and 48% for Python-Java translations. PPOCoder [PSR23] is a reinforcement learning-based code translation framework that incorporates CodeBLEU-inspired feedback during the training of the large language model. It employs a Boolean compiler feedback mechanism, which returns true if the generated code compiles and false otherwise.

TransCoder-ST [RLCL20] is an unsupervised code translation tool that leverages self-training and automated unit tests to ensure source-target code equivalence. It generates a synthetic dataset of equivalent code in two languages using these unit tests. However, the primary focus of these tools is to achieve logically correct translations, often at the expense of translation quality.

Chain-of-thought (CoT) has emerged as a groundbreaking tool in NLP, notably for its efficacy in complex reasoning tasks. CodeCoT [HBQC24] uses Chain-of-Thought approach to ensure the code their tool generates, follows the correct logic flow. Our CodeTransCoT approach also utilizes the Chain-of-thought approach to improve the complex code translation task.

Chapter 4

Challenges in Code Translation

In the past, Pan et. al. [PIK⁺24] have performed a large-scale evaluation and taxonomy of code translation bugs introduced by several general and code LLMs, when translating from a set of source languages to a set of target languages. They concluded that a large percentage of translation bugs can be attributed to the difference in syntax and semantics between the source and the target languages. Also, a large number of bugs are introduced due to the missing dependencies, incorrect logic implementation and incorrect handling of input - output data.

While it is useful to understand a general pool of translation bugs introduced by LLMs, we believe any approach to mitigate them, can be built only after classifying programming languages into various categories like strongly typed languages, loosely typed languages, object-oriented languages etc, based on the programming paradigm they support, and then, analyzing the broad syntactic and semantic distinctions between these source and target language categories. Pan et al. [PIK⁺24] observed that the effectiveness of code translation can vary significantly depending on the characteristics of the source and target programming languages, including factors such as the type system, available programming APIs, and support for metaprogramming through decorators or annotations. In our experiments also, we observed that understanding the differences in the programming paradigms from a given source code language to the required target code language is the key to produce more successful translations. *Why chose Java, Python* : In this paper, we chose the language pair (*Java, Python*) for code translation experiments as these two are representatives of two different programming paradigms. Java is an object-oriented language, while Python is a multi-paradigm language that supports object-oriented programming, procedural programming as well as functional programming. The fact that Java is strongly-typed and Python is loosely-typed poses a variety of challenges for LLMs when translating code from Java to Python and vice-versa. Many past work [PIK⁺24] have also experimented on this set of languages. Many good benchmarks like *AVATAR* [ATCC21] and *CODENET* [PKJ⁺21] are available for experimentation in this set.

Before we present our translation approach, in the next few subsections, we first discuss our key observations on the various bug categories introduced by LLMs, when translating from a more structured and strongly-typed language like Java to a more loosely-typed language like Python and vice-versa. The key observations we made during this investigation is what led us to our *TransCoT* approach.

The bug-categories we discuss below were generated as part of *vanilla prompting* experiments [PIK⁺24], where each prompt was of the following format.

```

1 $SOURCE_LANG:
2 // Unformatted source code
3 Translate the above $SOURCE_LANG code to $TARGET_LANG.
4 end with comment "<END-OF-CODE>".

```

SOURCE_LANG and *TARGET_LANG* were either *Java* or *Python*. We analyzed the translation results of around 500 standalone programs (250 Java programs and 250 Python programs) from *AVATAR* [ATCC21] benchmark, for which we had multiple test cases to account for a successful/unsuccessful translation. As in past work [RLCL20, PIK⁺24, JJJ⁺24], we deem a translation successful if it compiles, passes runtime checks, and all existing tests run successfully on the translated code. Furthermore, we go one step beyond a mere successful translation. We also consider the *Quality of Translation*, which is, how closely the generated code structurally resembles the original source code. We highlight why monitoring this quality of translation is important in the next few subsections

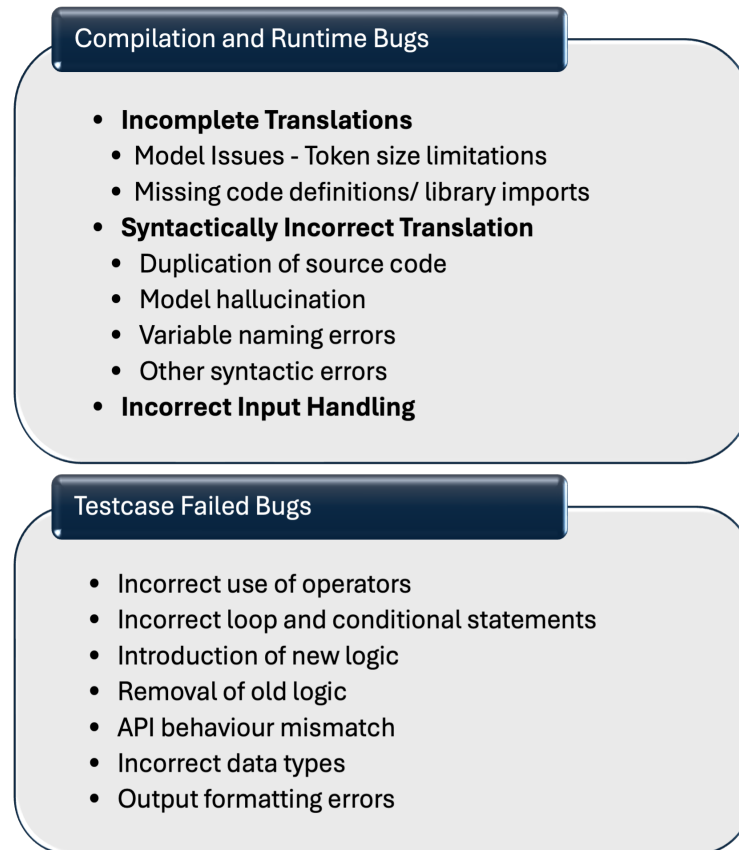


Figure 4.1: Categorisation of Translation Bugs Introduced by LLMs.

4.1 Bug Categories: Java \Rightarrow Python Translations

Python belongs to the category of loosely or dynamically typed programming languages and gives its programmers the flexibility to write functional code, procedural code, and even object-oriented code. In comparison, Java is an object-oriented and strongly typed language that requires proper data type declarations. These programming paradigm differences between these two languages make automatic translation of code from Java to Python (and vice-versa) a daunting task.

As shown in figure 4.1, we categorize the bugs introduced by LLMs during Java \Rightarrow Python translations into following two broad categories based on what time they manifest themselves - **Compilation or Runtime failed bugs**, and **Test case failed bugs**.

4.1.1 Compilation or Runtime Failed Bugs

Compilation bugs or Runtime failed bugs are where the translated code either fails to compile, or, breaks down before completing a full execution run. These result mostly from no or incomplete translations or syntactically incorrect translations.

Incomplete translations can happen due to model specific constraints like token size limitations. Missing code definitions and/or missing library imports in the translated code can break the code execution flow, thus generating compilation or runtime errors in the translated code.

Syntactically incorrect translations can happen when the model is unable to translate portions of source code and hence emits them as-is in the translated code. Another reason for syntactically incorrect translations is due to the hallucination problem in LLMs. Models may hallucinate sometime due to their limited contextual understanding of the given input source code.

Yet another reason for compilation or runtime-failed bugs is due to variable naming issues. An example of this is illustrated below.

```
1 // Original Java code
2 int[] [] range = new int[q][2];
3 ...
4 range[i][0] = sc.nextInt();

1 # Incorrect Translated Python Code
2 range = [list(map(int , input.split())) for _ in range(q)]
```

As shown, the variable name *range* is perfectly valid in the Java code, whereas in Python, it is the name of a commonly used built-in function and, therefore, does not qualify to be used as a variable name as it is causing an error when these built-in functions are called later in

the program. Other examples of such problematic variable names in Python are *max*, *min*, *in*, *out* etc.

Mismatch in input parsing format between the input source code and the translated code is another reason for a fairly significant number of runtime errors.

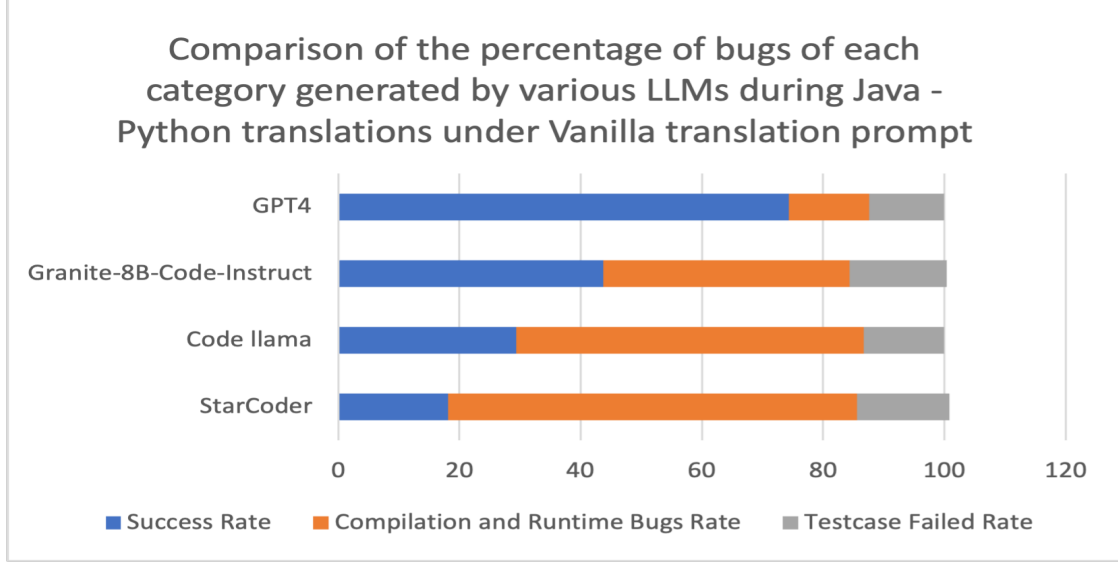


Figure 4.2: Comparison of percentage of bugs of each category in translated Python code. Benchmark: AVATAR.

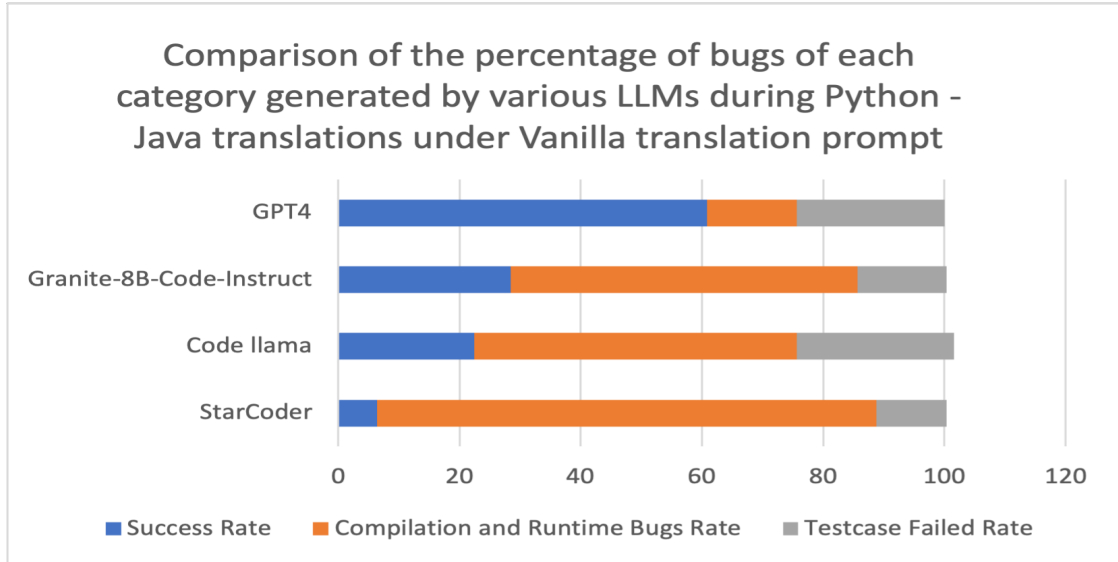


Figure 4.3: Comparison of percentage of bugs of each category in translated Java code. Benchmark: AVATAR.

We experimented with three open-source LLMs : *StarCoder* [LAZ⁺23] and *Code Llama* [?], and *Granite-8B-Code-Instruct* [MSZ⁺24] and one closed-source LLMs : *GPT-4* [OAA⁺23], to get an estimate of the percentage of bugs of each category, introduced by various LLMs, in

the translated Python and Java codes. Benchmark used in this experiment was *AVATAR* [ATCC21]. The results of the same are shown in figures 4.2 and 4.3.

Observation 1. Python programmers often structure solutions in the form of functions to write code that is shorter, easier to test, less prone to bugs, and well-suited for multiple applications. This functional programming paradigm considerably differs from Java’s object-oriented programming paradigm, and therefore results in a lot of compilation and run-time errors (figure 4.3) in the translated Java code. In the translated Java code, for open-source LLMs, between 53% - 82% bugs are compilation or runtime failed bugs. For closed-source LLMs also, this number is big, but is comparatively better placed at between 14% - 57%.

Observation 2. In the translated Python code, for open-source LLMs, between 57% - 67% bugs are compilation or runtime failed bugs. For closed-source LLMs, this number is between 13% - 40%.

Learning 1. During our experiments, we observed that if we instruct the LLMs to closely follow a well-structured, progressive, chain-of-thought based reasoning approach to translate code, it helps reduce the number of cases where the model may generate incomplete or syntactically incorrect translations, thereby considerably reducing a good number of compile-time errors. This learning forms the basis of our *TransCoT* approach.

4.1.2 Test Case Failed Bugs

Test case failed bugs are those that are due to logical errors in the translated code. Consider the example of a logical error shown below. In the translated Python code, the *XOR operator* from the original Java code is replaced with the *NotEqualTo operator*. Due to a mismatch in the precedence order of these two operators, an erroneous translation of the input conditional statement happens resulting in a logical error in the translated code.

```
1 if (a[m] >= 0 ^ a[i] >= 0) # Original Java code
2 if a[m] >= 0 != a[i] >= 0: # Incorrect Translated Python Code
```

Another example of a logical error is shown next. In the original Java source code, the loop index variable *i* is to be incremented within the loop body, if certain condition holds true. In the translated Python code, the normal loop formulation get replaced with a well-known Python built-in function called *range*. However, in Python, the *range* function prevents its loop index variable to be modified anywhere within the loop body, thereby breaking the code logic in this case.

```
1 // Original Java code
2 for (int i=0; i<input.length(); i++) {
3     // more code
4     if (count>max)
```

```

5     i++;
6     else
7         count++
8         // more code
9 }

```

```

1 # Incorrect Translated Python Code
2 for i in range(len(input)):
3     # more code
4     if count > max:
5         i+=1
6     else
7         count+=1
8     # more code

```

Many more examples of various reasons behind test case failed bugs were presented by Pan et. al. [PIK⁺24].

Observation 3. We observed that, while compilation and runtime failed bugs could be attributed to programming paradigm differences between Java and Python; the test case failed category of bugs occurred mostly due to language specific differences, e.g., API differences, between Java and Python. We also observed that prompt crafting techniques are not sufficient to resolve test case failed category of bugs, and that, these may only be resolved with more specific training of LLMs for it

4.2 Translation Quality Bugs

Compilation and Runtime failed bugs, and, Test case failed bugs are well discussed in all past work. We now introduce a new third category of translation bugs - *Translation Quality Bugs*.

A translation is deemed successful if it compiles, passes runtime checks, and all existing tests run successfully on the translated code. However, we claim that the story does not end here. To motivate why *translation quality* is important, consider the code translation example shown below.

As shown in figure 4.4, the original Java source code defines two nested classes: *BUnhappyHackingABCEdit* and *LightScanner*. The static member class *BUnhappyHackingABCEdit* defines a *solve* method, which first reads an input string from a given input stream, and then uses it to iteratively build and print an output string.

```

1 // Original Input Java Source Code
2 // Code Source - AVATAR Benchmark
3 public class XXX {
4     public static void main(String[] args) {
5         InputStream inputStream = System.in;
6         OutputStream outputStream = System.out;
7         LightScanner in = new LightScanner(inputStream);
8         PrintWriter out = new PrintWriter(outputStream);
9         BUnhappyHackingABCEdit solver=new BUnhappyHackingABCEdit();
10        solver.solve(1, in, out);
11        out.close();
12    }
13    static class BUnhappyHackingABCEdit {
14        public void solve(int testNumber, LightScanner in,
15                          PrintWriter out) {
16
17            String s = in.string();
18            StringBuilder d = new StringBuilder();
19            for (char c : s.toCharArray()) {
20                switch (c) {
21                    case '0':
22                        d.append("0"); break;
23                    case '1':
24                        d.append("1"); break;
25                    case 'B':
26                        if (d.length() > 0)
27                            d.setLength(d.length() - 1).trimToSize();
28                        break;
29                }
30            }
31            out.println(d);
32        }
33
34        static class LightScanner {
35            private BufferedReader reader = null;
36            private StringTokenizer tokenizer = null;
37
38            public LightScanner(InputStream in) {
39                reader = new BufferedReader(new InputStreamReader(in));
40            }
41
42            public String string() {
43                if (tokenizer == null || !tokenizer.hasMoreTokens()) {
44                    try {
45                        tokenizer = new StringTokenizer(reader.readLine());
46                    } catch (IOException e) {
47                        throw new UncheckedIOException(e);
48                    }
49                }
50                return tokenizer.nextToken();
51            }
52        }
53    }

```

Figure 4.4: Example of an input Java source code.

The other static member class is *LightScanner* class, which has methods to initialize an input stream and read an input string from it. The main class *XXX* defines a *main* method, which first instantiates an input stream using the *LightScanner* class, and then calls the *solve* method of class *BUnhappyHackingABCEdit*.

Next, consider the Python translation (figure 4.5) of the above Java code as generated by common LLM - *GPT-4*. The translation is syntactically and logically correct, and, it passes all test cases. However, the interesting point to note in the translated Python code is that, all class and subclass structure from the original Java source code has been scrapped, and, just the basic functionality of the program, i.e., to read an input string, and, use it to generate and print an output string has been put into a global method.

```

1 # Logically Correct Translation of above Java Code to Python Code
2 def solve():
3     s = input().strip()
4     d = []
5     for c in s:
6         if c == '0':
7             d.append('0')
8         elif c == '1':
9             d.append('1')
10        elif c == 'B':
11            if len(d)>0
12                d.pop()
13    print(''.join(d))
14 solve()

```

Figure 4.5: Python translation of Java code in figure 4.4, as generated by GPT-4 under vanilla prompting technique.

The above translation, although correct, raises a concern. What if the original Java program of figure 4.4 is a part of a larger Java package, which is to be entirely translated to Python. If in the package, there are some other Java classes, which instantiate and call the *solve* method of *BUnhappyHackingABCEdit* class, the connectivity between these classes would completely break after translation.

Take another example of a simple Python program (figure 4.6) this time. In the global scope of this program, is an integer variable named *a*, and a method named *func*, which computes and prints the sum of the digits of the integer variable *a*.

Next, consider the Java translation (figure 6.5) of the above Python code, as generated by common LLM - *StarCoder*, under vanilla prompt. The translation has compilation issues about missing imports, and, is also logically incorrect due to removal of last *if-else-statement block* in method *func*. However, the interesting point to note in the translated Java code is that, all method declarations and global variables declarations has been scrapped, and, just the basic functionality of the program has been put into a *main* method.

As in the previous example, the concern here also remains the same. What if the original

```

1 # Original Input Python Source Code
2 # Code Source - CodeNet Benchmark
3 a = int(input())
4 def func(a):
5     digits = []
6     temp = a
7     while temp != 0:
8         digits.append(temp%10)
9         temp = int(temp/10)
10    result = sum(digits)
11    if result == 1:
12        print('10')
13    else:
14        print(result)
15 func(a)

```

Figure 4.6: Example of an input Python source code.

Python program of figure 4.6 is a part of a larger Python package, which is to be entirely translated to Java. If in the package, there are some other Python programs, which call the *func* method and/or access the global variable *a*, the connectivity between these programs would completely break after this translation.

```

1 // Logically Incorrect Java Translation of above Python Code
2 // Code also has Compilation Errors
3 public class YYY{
4     public static void main(String[] args) {
5         Scanner sc = new Scanner(System.in);
6         int a = sc.nextInt();
7         int sum = 0;
8         while(a!= 0) {
9             sum += a % 10;
10            a /= 10;
11        }
12        System.out.println(sum);
13    }
14 }

```

Figure 4.7: Java translation of Python code in figure 4.6, as generated by StarCoder under vanilla prompting technique.

An important question now to ask here is -

What is the purpose of using LLMs for code translation? Do we need them to translate standalone programs or entire legacy code bases?

If standalone programs be the answer, we are great with what we are seeing at this level, i.e., more syntactically and logically correct translations. However, we believe the answer to the above question is legacy code bases, as standalone programs may well be translated manually rather than spending so many model resources on them. Now let us see what comprises a code base? Indeed it is a set of interconnected programs, and not just a bunch of standalone programs. This brings us to an important research question.

RQ : How can entire code bases, which are essentially a large set of inter-

connected programs, be effectively translated from one programming language to another, without disturbing their interconnections?

The answer to the above question is - by as far as possible, preserving the original program structure even after translating to the target language.

Learning 2. During our experiments, we observed that, LLMs produce better quality translations, if we instruct them to follow a chain-of-thought based reasoning approach in which they first understand the original source code structure, and then progressively translate portions of source code while still preserving the original structure in the output code. This learning also forms the basis of our *TransCoT* approach

Since *quality of translation* is an important parameter, for our experiments, we use it as another metric to measure the translation quality of LLMs.

Chapter 5

METHODOLOGY

In the previous Chapter, we have seen different types of bugs and challenges associated with the code translation task. The motivation behind our approach is to deal with challenges faced by LLMs in code translation. Based on the observations and learnings in Chapter 4, some of the main things to focus on while working with LLMs are to make them understand what task we want to solve. Another thing is to assist them in solving the task without getting hallucinated. These things can be achievable with LLMs by providing them with relevant information and descriptions of the problem task with appropriate approaches. To make LLMs work for any task, we can interact with LLMs by assuming them as black boxes, with the help of prompting instructions to provide input information and decoding-based strategies for generating output. We have shown the evaluation part and output generation part of LLMs in Chapter 7. So, we can mainly formulate approaches for LLMs by encoding relevant information in the form of prompts or instructions.

5.1 Vanilla Approach

One of the naive ways to give a description of the problem task to LLMs is a simple vanilla prompt. For this code translation task, the vanilla prompt provides simple instructions for translating the source language code to the target language code. Figure 5.1 has a vanilla prompting template for any LLM, but it can differ slightly according to different LLMs based on specific tokens and generation styles.

```
$SOURCE_LANG:
$SOURCE_CODE
// Unformatted source code
Translate the above $SOURCE_LANG code
to $TARGET_LANG end with comment
" <END-OF-CODE>".
$TARGET_LANG:
// Code generated
```

Figure 5.1: Vanilla prompting template.

5.1.1 Discussions

The Vanilla Approach is a really simple approach that has faced a low translation rate and different types of bugs. In Figure 4.2 and Figure 4.3, we can see that a significant portion of errors are runtime and compilation errors. Specifically, it constitutes more than 50% failing rate in most open source LLMs. One of the main reasons is that LLMs are not able to figure out relevant information for code translation itself, hence resulting in significant errors due to hallucination. So there is a need to provide additional contextual information in Prompt based approaches.

5.2 Additional Context based Approach

In addition to simple prompting instruction we can also provide relevant additional information to make easy for LLM to do code translation of *complex programs*. Additional information for programs in source code language in dataset, can be sample INPUT/OUTPUT format information, explanation of source language code in nature language description, program analysis information extracted out of source code language such as type information, Abstract Syntax Tree(AST) related information, etc.

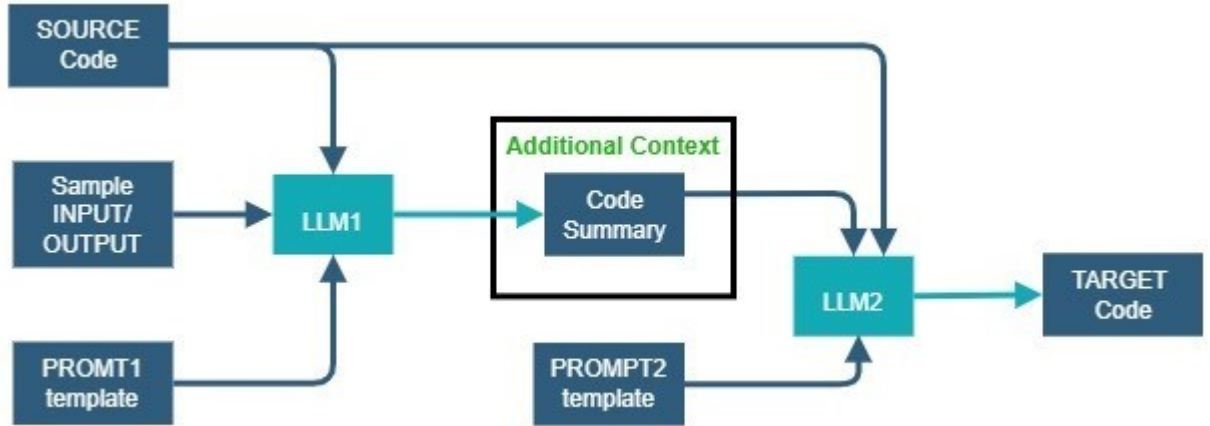


Figure 5.2: Additional Context-based Approach

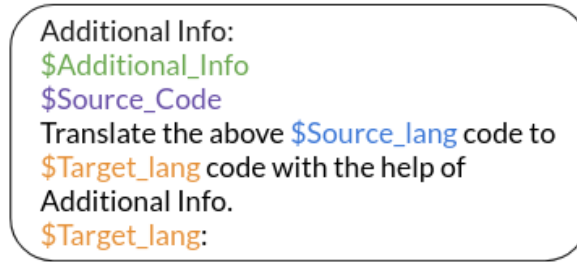
5.2.1 Type information based Approach

Providing type information of the source code as additional context helps LLMs understand the variables and their associated types, thereby reducing semantic errors in the generated program. This is particularly crucial when the source language is dynamically typed, such as Python, and the target language is statically typed, such as Java. However, extracting

type information from dynamically typed languages poses a challenge. To address this, we employ deep learning approaches to infer variable types in the source code. One such tool is HiTyper [PGL⁺22], a hybrid type inference tool built on the Type Dependency Graph (TDG). Given that these approaches can be prone to errors, we apply them only to unsuccessful translations of the vanilla approach.

More Details

Here is the prompting template for this approach in Figure 5.3, which is used in the Additional Context-based approach’s pipeline depicted in Figure 5.2. In this pipeline, we first infer type information using HiTyper and then feed it in as natural language to prompt shown in Figure 5.3. Finally, we will generate a target language code by giving the resultant prompt to LLM, which is a StarCoder model [LAZ⁺23] for this experiment.



Additional Info:
 \$Additional_Info
 \$Source_Code
 Translate the above \$Source_lang code to
 \$Target_lang code with the help of
 Additional Info.
 \$Target_lang:

Figure 5.3: Type information based prompt

5.2.2 Dynamic Function Trace

This is a runtime based approach where we select a test case and incorporate dynamic function trace data as supplementary information for elicitation.

This process operates at two distinct levels:

- At the initial level, we produce outcomes employing the conventional (Vanilla) prompt method. Specifically, we select source code samples that result in either a testcase fail or runtime fail error in the Vanilla prompt technique.
- Subsequently, we capture the call sequence details for both the source and the resulting target code, integrating them as supplementary information. We then task the Language Model (LLM) with regenerating a response based on this augmented input.

The testcases are already present in the dataset we use i.e Avatar [ATCC21] and CodeNet [PKJ⁺21]. We use ufrace function call tracer library for finding out the dynamic call sequence.

More Details

The Prompting template for this approach is shown in 5.4. We first extract out the call sequence for a given testcase using ufttrace module. We then feed in the call sequence as input in natural language and make it as simple as possible so that it is easier to understand. As we can see in the prompting template, we first describe the structure of the call sequence and then only proceed to feed in the additional dynamic info.

Let sequence of calls be defined as:
 x -> y represents that y was called after x
 x and y are represented by <class-name>.<method-name>
 x(n) tells that x was called n times contiguously

The `$Source_lang` code:
`$Source_code`

The sequence of calls for the above code :
`$Source_code_call_sequence`

Your generated `$Target_lang` code:
`$Target_code`

The sequence of calls for the above code:
`$Target_lang_call_sequence`

Can you re-generate your response and correct the above `$Target_lang` code. Do not add any natural language description in your response.
`$Target_lang` Code:

Figure 5.4: Dynamic Function Trace Prompt

Before proceeding please refer Lalit's CodeTransCoT based approaches [Mee24] as the approaches mentioned from here are improvements to his CodeTransCoT technique.

5.3 Program analysis based approach

We have observed that CodeTransCoT performs exceptionally well on the Avatar and CodeNet datasets. The main focus of the CoT method is to reason through and generate a structurally similar program to the source code, which helps retain logic and reduce compilation errors in the target language. The first step in the CodeTransCoT reasoning task is to infer the structure of the source code. Instead of relying on LLMs to generate this structure, we can perform Abstract Syntax Tree (AST) analysis to extract the structure and provide it as the first step in the prompt. This approach eliminates errors that can arise when LLMs struggle to determine the structure due to the complex nature of the source code.

5.3.1 More Details

For implementing this we will be using the Tree Sitter library. Tree-sitter is an incremental parsing library and an incremental parsing system that generates parsers for programming languages. It is used for parsing source code and generating a parse tree, which can be used to understand the structure of the code. We extract the Step-1 of the CodeTransCoT approach and feed it into the LLM as shown in the prompt below.

Prompt template Java-Python Program analysis

####

1. Java Code:

```
// <<Source_Code in Java>>
import <<requirements>>
public class Main {
    //<<class variables declarations>>
    public static void main ( String [ ] args ){
        //<<"main" function definition>>
    }
    static class CustomScannerClass {
        //<<class variables declarations>>
        String next ( ){
            //<<"next" function definition>>
        }
        long otherFunctions ( ){
            //<<"otherFunctions" function definition>>
        }
    }
    static class HelperClasses {
        //<<class variables declarations>>
        HelperClasses(){
            //<<class variables initialization and other statements>>
        }
        String helper ( ){
            //<<"helper" function definition>>
        }
    }
}
```

1. Sample Input:

```
// <<Sample Input Test case>>
```

1. Expected Output:

```
// <<Expected Output of given Sample Input Test case>>
```

1. Steps: Let's think step by step.

Step 1: First of all identify classes and functions declarations present in the above Java Code-
Identified classes and functions declarations in the Java code:


```

public class Main {
    public static void main ( String [ ] args );
    static class CustomScannerClass {
        String next ( );
        long otherFunctions ( );
    }
    static class HelperClasses {
        HelperClasses();
        String helper ( );
    }
}

```

Step 2: Considering Step 1, classes and functions declarations should be presented in the corresponding Python Code-

Identified classes and functions declarations in the Python code:

Class Main:

```

def main(self):

class CustomScannerClass:
    def __init__(self):

    def next(self):

    def otherFunctions(self):

class HelperClasses:
    def __init__(self):

    def helper(self):

```

Step 3: Complete translation of the above Java code to Python code by considering identified classes and functions declarations in Step 2. Consider above sample input and output format information to handle input and output related code properly in the generated Python code. While generating the Python code, keep track of required library imports to be added in the Python Code. Make sure your generated code is syntactically correct-

1. Python Code:

```

import sys
Class Main:
    def main(self):
        #<<"main" function definition>>

class CustomScannerClass:
    def __init__(self):
        #<<some initializing statements>>
    def next(self):
        #<<"next" function definition>>
    def otherFunctions(self):
        #<<"otherFunctions" function definition>>

```

```

class HelperClasses:
    def __init__(self):
        #<<some initializing statements>>
    def helper(self):
        #<<"helper" function definition>>

if __name__ == "__main__":
    obj = Main()
    obj.main()

####$###
2. Java Code:
<<2nd demonstration example>>
Similarly there can be other demonstration example for the In-context learning (ICL).
In our approaches, we have used two demonstration examples.

3. Java Code:

####$###
// <<$Source_Code of new Test program in Java>>

3. Sample Input:

// <<Sample Input Test case for this new Test program>>

3. Expected Output:

// <<Expected Output of this Sample Input Test case>>

3. Steps: Let's think step by step.

<<Extract Step-1 from Tree Sitter >>
<<LLM will generate Step-2 from here onwards. >>

```

5.4 Iterative Improvement

In this section, we explore an iterative approach methodology. The main drawback of CodeTransCoT is its reliance on fixed demonstrations for every task. We have observed that this leads to bias in the translated code, as it tends to use functions or classes from the demonstrations even when unnecessary, thereby affecting translation quality.

To mitigate this issue, we propose using a sampling technique to select demonstrations based on the structure of the source code. By doing so, we can alleviate bias while preserving the logic and structure of the source code. Initially, we will maintain a database of k -demonstrations. For each task, we will sample 2 out of the k demonstrations using TF-IDF

similarity. After running an iteration, we can retain the demonstrations generated by the LLM, add them back to the database, and run another iteration for those that were evaluated as incorrect. This process increases the diversity of demonstrations in the database, which can improve the execution accuracy of the samples, leading to better results after every iteration. The workflow is shown in Figure 5.5

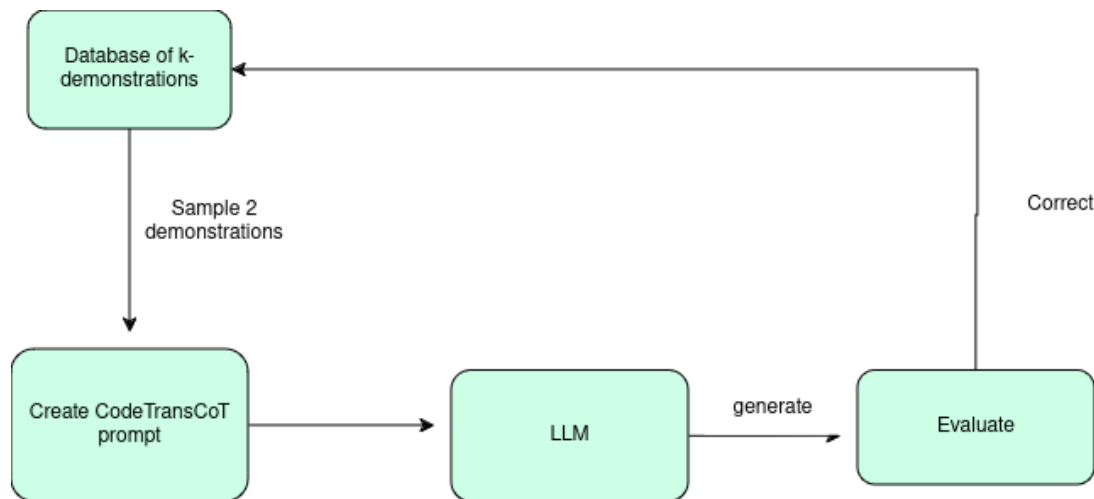


Figure 5.5: Workflow of the Iterative prompt approach

Chapter 6

Evaluation Metrics

Large Language Models (LLMs) have shown exceptional proficiency in generating text that closely resembles human writing, and their applications now extend to code translation. However, evaluating the quality of the generated code presents a unique set of challenges. Traditional metrics like the BLEU score are match-based metrics requiring a reference solution and are also unable to capture complex space of functionally equivalent translations.

This chapter addresses these challenges and introduces the evaluation metrics used. We use the unbiased estimator $\text{pass}@k$ [CTJ⁺21] to evaluate our translated code for functional correctness with the help of the test cases. We also introduce a new metric, **Quality Metric**, to assess structural quality correctness that aligns with our CoT approach.

6.1 Pass@ k estimator

The $\text{pass}@k$ metric is defined as the probability that at least one of the top k -generated code samples for a problem passes the unit tests.

For a given task, $n \geq k$ samples are sampled. Let the number of correct samples be denoted by c , among the n samples, which pass all the tests. The probability of sampling only incorrect generated code as top- k samples is $\binom{n-c}{k} / \binom{n}{k}$.

Then $\text{pass}@k$ is defined as :

$$\text{pass}@k = \mathbb{E}_{\text{problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]$$

Direct computation of this estimator is numerically unstable. We use the numerically stable numpy implementation introduced by [CTJ⁺21].

For a special case when $k = 1$, $\text{pass}@1$ evaluates down to:

$$\begin{aligned} \text{pass}@1 &= \mathbb{E}_{\text{problems}} \left[1 - \frac{n-c}{n} \right] \\ &= \mathbb{E}_{\text{problems}} \left[\frac{c}{n} \right] \end{aligned}$$

But if greedy decoding is used, the number of correct samples c is either $c = n$ or $c = 0$. Let P be the set of all the programs in the target language which are functionally equivalent

to the task at hand and x is the generated program. Then we define pass@1^* ,

$$\text{pass@1}^* = \mathbb{E}_{\text{problems}} [\mathbb{1}\{I_P(x) = 1\}]$$

where $I_P(x)$ is an indicator function.

6.2 Quality Metric

Given the reliance of our Cot method on identifying structural components in the source language and converting them into an analogous structure in the target language, it is essential to assess the quality of the translation to guarantee and verify the reasoning steps and make sure we have indeed generated what we have expected.

Hence, we introduce a Structural Quality assessment measure that evaluates the similarity between Static call graphs derived from both the source code and the translated code by employing graph edit distance as the evaluation criterion

<pre> 1 // Java Code 2 public class Main(){ 3 Main() { 4 //some code 5 } 6 long calc() { 7 return dfs(arg1, arg2); 8 } 9 boolean isOK(arg1) { 10 //some code 11 } 12 long dfs(arg1, arg2) { 13 //some code 14 isOK(arg1); 15 dfs(arg1, arg2); 16 } 17 public static void main(String[] args) { 18 Main ins = new Main(); 19 System.out.println(ins.calc()); 20 } 21 }</pre>	<pre> 1 # Python Code 2 class Main: 3 4 def __init__(self): 5 #some code 6 7 def calc(self): 8 return self.dfs(arg1, arg2) 9 10 def isOK(self, arg1): 11 #some code 12 13 def dfs(self, arg1, arg2): 14 #some code 15 self.isOK(arg1); 16 self.dfs(arg1, arg2) 17 #some code 18 19 if __name__ == "__main__": 20 ins = Main() 21 ins.calc()</pre>
--	--

Figure 6.1: Comparison of structural representations between a Java source code and its Python translation using the Vanilla approach through the StarCoder LLM.

6.2.1 Call Graph

A call graph is a control-flow graph that depicts the calling relationships between subroutines in a program. In this graph, each node represents a procedure, and each edge (f, g) signifies

that procedure f calls procedure g . Consequently, a cycle in the graph indicates recursive procedure calls.

For example, consider two functionally equivalent programs in two languages (Java and Python), as shown in Figure 6.1. The Python program was obtained by translation using vanilla prompting technique on the source Java Code with the help of StarCoder[LAZ⁺23]. The corresponding Call Graphs for the two programs are given in Figure 6.2.

It is evident that there needs to be a better match in the call graphs of the two programs. There needs to be a `Main.main` function in the Python program, which is supposed to call `Main.calc`. Even though the programs run correctly without errors, the source code's function calls and definition structure are not preserved in the generated code.

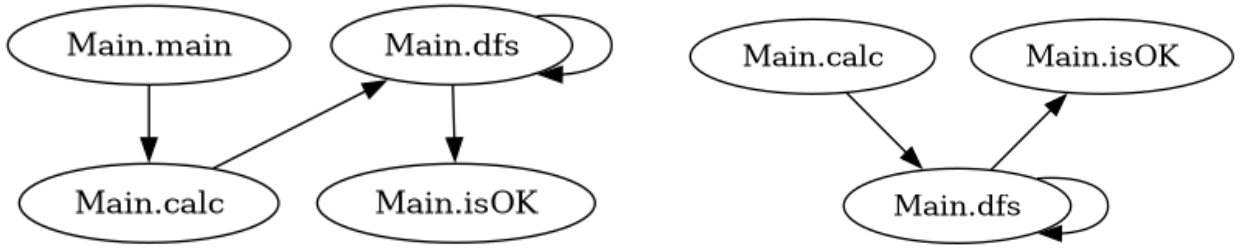


Figure 6.2: Corresponding Java call graph (left) and Python call graph (right) of the Code in 6.1

6.2.2 Graph Edit Distance

To compare the static call graphs in Figure 6.2, we need a function which takes inputs as two graphs and returns a metric indicating the similarity between the two graphs.

Generally, given a set of graph edit operations (also known as elementary graph operations), the graph edit distance between two graphs g_1 and g_2 , denoted as $GED(g_1, g_2)$ can be defined as,

$$GED(g_1, g_2) = \min_{(e_1, \dots, e_k) \in \mathcal{P}(g_1, g_2)} \sum_{i=1}^k c(e_i)$$

where $\mathcal{P}(g_1, g_2)$ denotes the set of edit paths transforming g_1 into (a graph isomorphic to) g_2 and $c(e) \geq 0$ is the cost of each graph edit operation e . For simplicity, We use a cost of 1 for each operation in all our experiments.

The set of elementary graph edit operations includes:

- Vertex insertion: Introducing a new labeled vertex to the graph.
- Vertex deletion: Removing a vertex from the graph.

- Edge insertion: Adding a new edge between a pair of vertices.
- Edge deletion: Removing an edge between a pair of vertices.

The elementary operations correcting the Python call graph in Figure 6.2 can be visualized in Figure 6.3. The reference program is also given in Figure 6.4. We add a new node `Main.main` and a new edge connecting `Main.main` and `Main.calc` giving us a cost of 2. It can be proved that this is the minimum-cost edit path transforming the Python call graph.

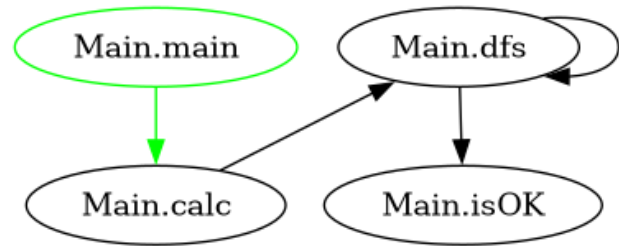


Figure 6.3: Correcting the call graph

```

# Python Code
class Main:

    def __init__(self):
        #some code

    def calc(self):
        return self.dfs(arg1, arg2)

    def isOK(self, arg1):
        #some code

    def dfs(self, arg1, arg2):
        #some code
        self.isOK(arg1):
        self.dfs(arg1, arg2)
        #some code

    def main(self):
        #some code
        self.calc()

if __name__ == "__main__":
    ins = Main()
    ins.main()
  
```

Figure 6.4: Reference program for the source Java program in Figure 6.2

Creating a reference Java program while translating from Python to Java is a bit tricky. For example, the reference for the program given in Figure 6.5 is shown in Figure ???. For more details on the implementation refer chapter 7

```
1 import java.util.*;
2 import java.io.*;
3 import java.util.stream.Collectors;
4
5 public class YYY{
6     static int a;
7
8     public static void func(int a) {
9         int digits[] = new int[100];
10        int temp = a;
11        int i = 0;
12        while (temp!= 0) {
13            digits[i] = temp % 10;
14            temp = temp / 10;
15            i++;
16        }
17        int result = Arrays.stream(digits).sum();
18        if (result == 1) {
19            System.out.println("10");
20        } else {
21            System.out.println(result);
22        }
23    }
24
25    public static void main(String[] args) {
26        Scanner scanner = new Scanner(System.in);
27        a = scanner.nextInt();
28        func(a);
29    }
30 }
```

Figure 6.5: Reference Java program for quality analysis of Python code in figure 4.6

Chapter 7

Experiments

This chapter presents a detailed account of the experimental setup, methodologies, and procedures employed to answer the research question. Subsequent subsections outline the models used, the evaluation results of the approaches mentioned in preceding chapters, and the metrics mentioned in (chapter 6). We have used 1x A100-Nvidia GPUs to perform all experiments unless stated otherwise.

7.1 Models

StarCoder [LAZ⁺23]: A pre-trained LLM of 15.5B parameters with 8K context length and infilling capabilities. It is trained on 1 trillion tokens sourced from The Stack [KLBA⁺22], a comprehensive collection of permissively licensed GitHub repositories, and further fine-tuned with 35 billion Python tokens.

GPT-4 [OAA⁺23]: We use the OpenAI API to access the gpt-4 model, which has a context window of 8,192 tokens and uses training data up to Sep 2021.

Code Llama [RGG⁺24]: An open-source large language model for code derived from Llama 2, offering state-of-the-art performance, infilling capabilities, and support for large input contexts. Specifically, we use the Code Llama-Instruct 13B model, which has been instructionally fine-tuned after being pre-trained on 500 billion tokens from a code-centric dataset.

IBM-Granite Code Models [MSZ⁺24]: A family of decoder-only code models designed for code generation tasks, trained on a diverse corpus encompassing code written in 116 programming languages. It consists of models ranging in size from 3 to 34 billion parameters. We use the Granite-8B-Code-Instruct and Granite-20B-Code-Instruct for our experiments, where the Code-Instruct models is fine-tuned using a combination of Git commits paired with human instructions and open-source synthetically generated code instruction datasets.

7.2 Datasets

CodeNet [PKJ⁺21] is a comprehensive collection of code samples accompanied by extensive metadata. It is sourced from two online judge websites: AIZU and AtCoder, which present

programming challenges through courses and contests. CodeNet encompasses 13,916,868 submissions across 4,053 unique problems. The submissions are written in 55 different programming languages, with the majority (95%) utilizing C++, Python, Java, C, Ruby, and C#. We randomly sample 250 problems of Java and Python among the 4053 problems and further sample one correctly submitted program for each problem along with its testcase.

Avatar [ATCC21] (jAVA-pyThon progrAm tRanslation) includes solutions written in Java and Python for 9,515 programming problems gathered from competitive programming sites like Codeforces, Google Code Jam, AIZU Online Judge, AtCoder, and from online platforms such as LeetCode, GeeksforGeeks, Project Euler, alongside contributions from open-source repositories. AVATAR contains unit tests for 250 evaluation examples to perform functional accuracy evaluation of the translation models. The unit tests are gathered from publicly available test cases published by AtCoder.

7.3 Results

Table 7.1: Results of the various approaches mentioned in Methodology 5

Dataset	Source	Target	Approach	Models	
				StarCoder	Code Llama
Avatar	Java	Python	Vanilla	18.1	29.2
			TypeInfo	21.2	32.4
			DynamicTrace	20.3	30.7
			ProgAnalysis	46.6	37.3
	Python	Java	Vanilla	6.4	22.4
			TypeInfo	13.4	20.9
			DynamicTrace	11.4	21.4
			ProgAnalysis	29.3	27.1

Code Llama generally performs better than StarCoder in the baseline (Vanilla) and TypeInfo approaches. However, ProgAnalysis significantly boosts StarCoder’s performance, making it the best-performing model for both translation directions when using this approach. We can see that DynamicTrace sometimes performs worse than TypeInfo approaches which might be due to prompt being too complex for these LLMs to understand, as these LLMs are not powerful enough for large context window prompts.

Table 7.2 shows the results for Iterative method. We use $k = 5$ for all settings. We can see that the results improve for both the models after the first iteration. There is a 5% increase in the Granite-8B model which is even better than CodeTransCoT approach

Iteration	Models	
	StarCoder	Granite-8B
Iter-1	0.37	0.43
Iter-2	0.4	0.48

Table 7.2: Iterative technique results

[Mee24] which gives 44% as Pass@1 accuracy for the same setting.

7.3.1 Evaluations of the Code Translation Quality

Table 7.3: Quality metric results for the Vanilla and CodeTransCoT approach for all the models. The reported number is average graph edit distance across the whole dataset

Dataset	Source	Target	Approach	Models			
				StarCoder	Code Llama	Granite-8B	Granite-20B
Avatar	Java	Python	Vanilla	3.40	3.72	3.86	2.94
			CodeTransCoT	2.84	1.35	2.81	2.86
	Python	Java	Vanilla	2.66	2.35	2.34	2.70
			CodeTransCoT	2.01	2.18	2.19	1.92
CodeNet	Java	Python	Vanilla	7.17	7.34	7.30	7.01
			CodeTransCoT	6.89	5.80	6.30	6.17
	Python	Java	Vanilla	1.74	1.31	1.26	1.02
			CodeTransCoT	1.15	0.98	1.03	1.05

Table 7.3 shows the translation quality metric (refer to chapter 6) for all the settings. Since we follow Chain-of-Thought to generate a structured program similar to the source program, we expect the average graph distance computed to be lower in the CodeTransCoT approach than when computed with the Vanilla approach.

We would need a reference call graph and the generated program’s call graph to compute the graph edit distance. Some examples for obtaining such graphs are shown in chapter 6.

Some implementation steps we followed are:

1. We do not consider library calls as nodes in the graph and only consider user-defined functions and classes and the interconnections between them, forming the edges.
2. To simplify it, We removed any implicit/explicit init or constructors.
3. Even though computing graph edit distance is an NP-hard problem, the number of nodes in the source graph does not exceed 20, which makes it inexpensive.
4. We used Java-callgraph [Gou11] and PyCG - Practical Python Call Graphs [SSL⁺21] to extract Java and Python call graphs, respectively

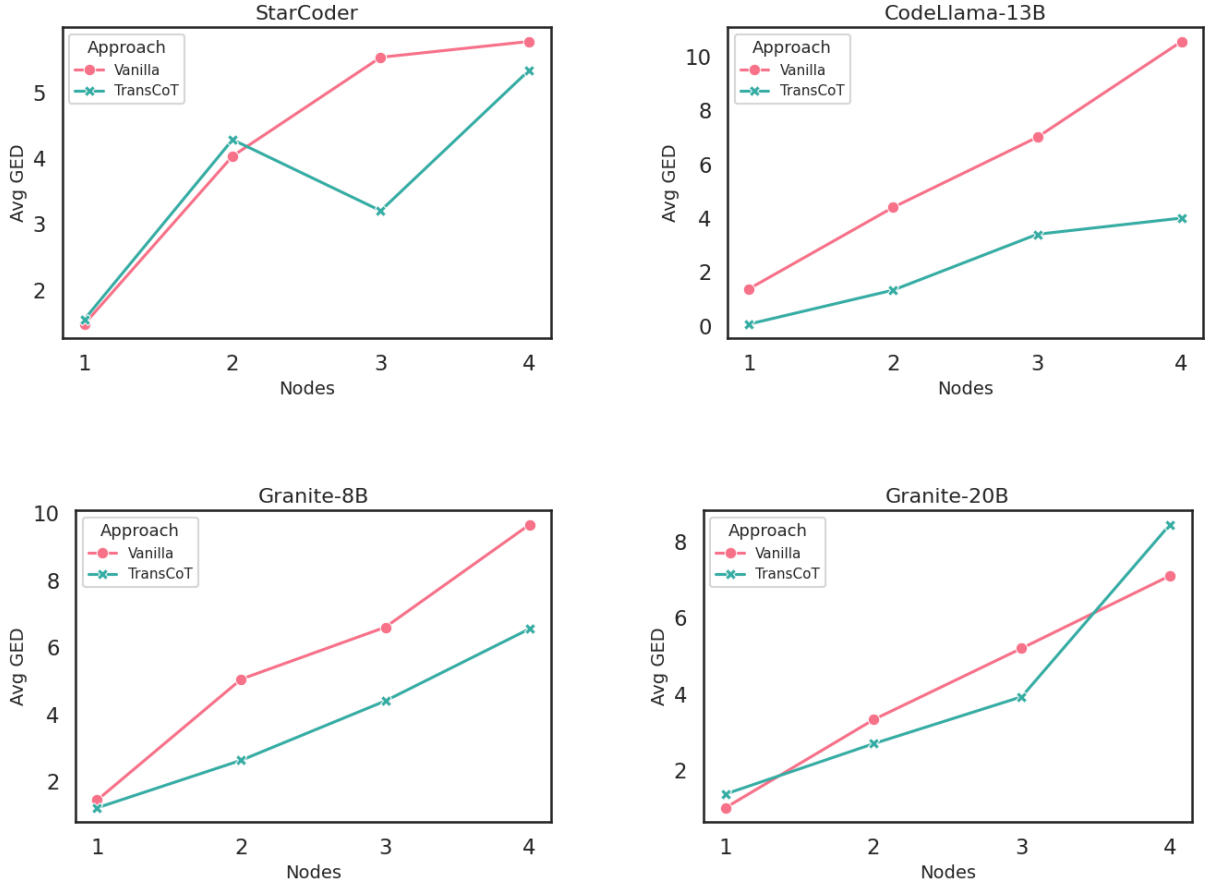


Figure 7.1: Number of nodes vs Average graph edit distance plots. This is on Avatar dataset and translation was from Python to Java

The datasets we have used contain programs from competitive programming platforms, which means that the program’s complexity and structure vary widely on the problem and the user writing the program. So, there is a need to classify and analyze the problems in the dataset based on structural complexity. For this reason, we break the dataset into different categories based on the number of nodes in the program’s call graph. Figure 7.1 Shows some line plots where, on the x-axis, we plot the number of nodes, and on the y-axis, we plot the average graph edit distance values.

Figure 7.1 shows the plots for all the models while translating from Java to Python, and for Python to Java, the plots are shown in Figure 7.2. Firstly, there seems to be a positive correlation between the number of nodes and the quality metric value for both methods. In other words, the more complex the source program gets, the harder it is to maintain structure while translating. Secondly, CodeTransCoT does show better performance regarding quality metrics but is only marginally better. This is mainly due to the bias in CoT where the functions or classes in the In-Context demonstrations are added to the target program when it was not necessary, thus reducing the measure of the quality metric.

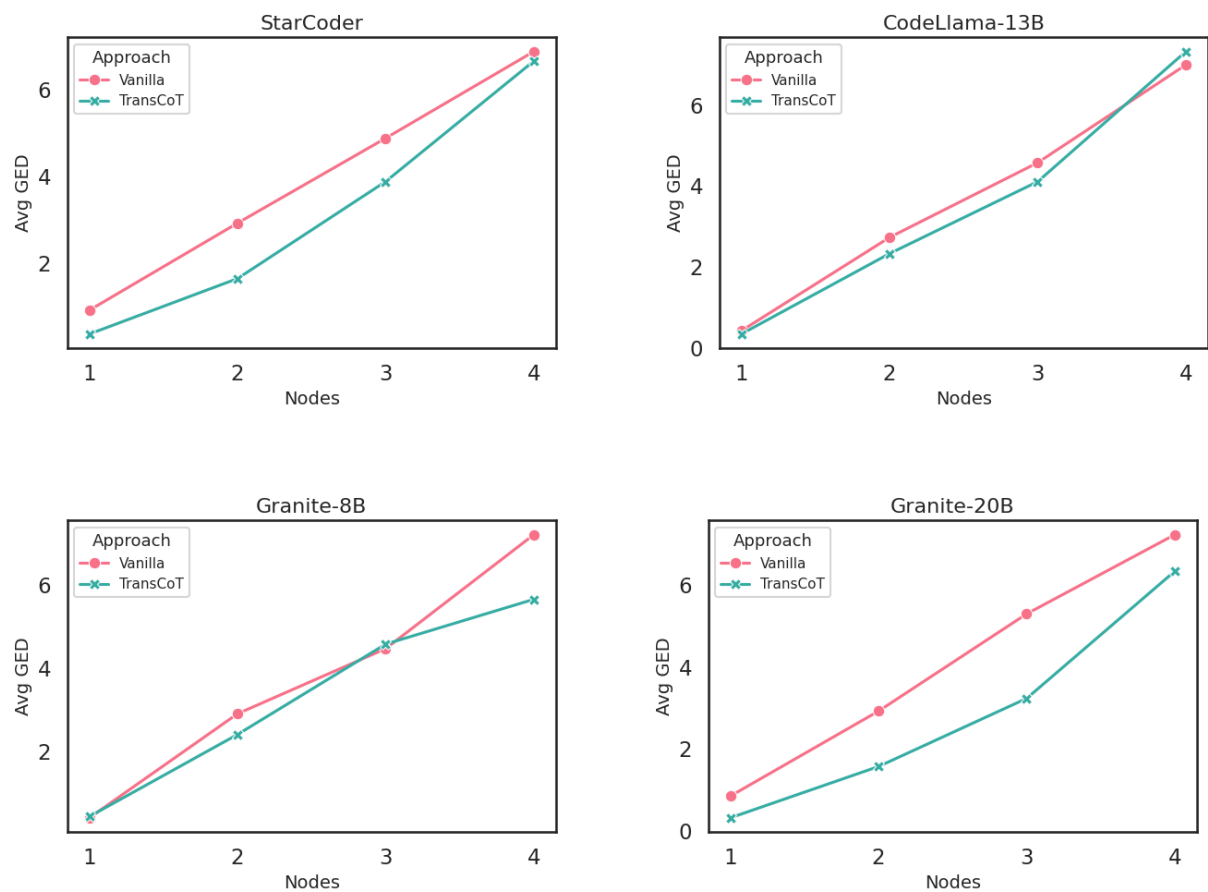


Figure 7.2: Number of nodes vs Average graph edit distance plots. This is on Avatar dataset and translation was from Java to Python

Chapter 8

Conclusion

8.1 Conclusion

We have presented an approach for evaluating structural quality of the code translation task. We also examine providing type information or runtime information to the LLM to aid in translation. Observing the success of CodeTransCoT, we also propose an iterative approach which helps in further improving the results in an iterative manner. We verify CodeTransCot generations using the Quality metric which incorporates static call graph analysis and computing graph edit distance to evaluate the difference in structure of two programs.

Reproducibility Statement. To ensure the reproducibility of our proposed work, we have included prompting templates for each approach. Detailed generation settings are provided in the Experiments Chapter, with additional information on crafting the prompts, including examples, available in the Appendix. The code of the proposed method and related artifacts will be released after the acceptance of the research paper based on this work.

Bibliography

- [ATCC21] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590*, 2021.
- [CTJ⁺21] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.
- [Fom19] Nikita Fomin. py2java: Python to java language translator, 2019. <https://pypi.org/project/py2java/>, 2019.
- [Gou11] Georgios Gousios. Java-callgraph: Java call graph utilities. <https://github.com/gousiosg/java-callgraph>, 2011.
- [HBQC24] Dong Huang, Qingwen Bu, Yuhao Qing, and Heming Cui. Codecot: Tackling code syntax errors in cot reasoning for code generation, 2024.
- [JJJ⁺24] Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. Cotran: An llm-based code translator using reinforcement learning with feedback from compiler and symbolic execution, 2024.
- [KLBA⁺22] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code. *Preprint*, 2022.
- [LAZ⁺23] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim,

- Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.
- [LRCL20] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages, 2020.
- [Mee24] Lalit Meena. Codetranscot: Enhancing quality of code translation with structured chain-of-thought prompting. Master’s thesis, Indian Institute of Technology Delhi, 2024. https://github.com/shashank-g12/Code-Translation/blob/main/MTP_these/Lalit_thesis.pdf.
- [MSZ⁺24] Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, Manish Sethi, Xuan-Hong Dang, Pengyuan Li, Kun-Lung Wu, Syed Zawad, Andrew Coleman, Matthew White, Mark Lewis, Raju Pavuluri, Yan Koyfman, Boris Lublinsky, Maximilien de Bayser, Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Yi Zhou, Chris Johnson, Aanchal Goyal, Hima Patel, Yousaf Shah, Petros Zerfos, Heiko Ludwig, Asim Munawar, Maxwell Crouse, Pavan Kapanipathi, Shweta Salaria, Bob Calio, Sophia Wen, Seetharami Seelam, Brian Belgodere, Carlos Fonseca, Amith Singhee, Nirmal Desai, David D. Cox, Ruchir Puri, and Rameswar Panda. Granite code models: A family of open foundation models for code intelligence, 2024.
- [OAA⁺23] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael,

Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rameesh Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O’Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorný, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil

- Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2023.
- [PGL⁺22] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Static inference meets deep learning: a hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering, ICSE '22*. ACM, May 2022.
- [PIK⁺24] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24*. ACM, April 2024.
- [PKJ⁺21] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 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)*, 2021.
- [PSR23] Sindhu Tipirneni Parshin Shojaee, Aneesh Jain and Chandan K Reddy. Execution-based code generation using deep reinforcement learning. arxiv preprint arxiv:2301.13816, 2023, 2023.
- [RGG⁺24] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.
- [RLCL20] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33:20601–20611, 2020.

- [SRL⁺23] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. Code translation with compiler representations, 2023.
- [SSL⁺21] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. Pycg: Practical call graph generation in python, 2021.
- [TMS16] Ling Li Iulius Curt Troy Melhase, Brian Kearns and Shyam Saladi. java2python: Simple but effective tool to translate java source code into python. <https://github.com/natural/java2python>, 2016.
- [tss23] Tss. the most accurate and reliable source code converters, 2023.(tangible software solutions). <https://www.tangiblesoftwareolutions.com>, 2023.
- [XNFR23] Yiqing Xie, Atharva Naik, Daniel Fried, and Carolyn Rose. Data augmentation for code translation with comparable corpora and multiple references, 2023.