TranSCoT: Towards Improving Quality of Code Translations

Lalit Meena*
lkmjeel.19@gmail.com
Indian Institute of Technology
New Delhi, India

Shashank Govindappa*
shashankgovindappa110@gmail.com
Indian Institute of Technology
New Delhi, India

Monika Gupta
monika.gupta.0878@gmail.com
Indian Institute of Technology
New Delhi, India

Anamitra Choudhury
anamchou@in.ibm.com
IBM Research
New Delhi, India

Vijay Arya
vijay.arya@in.ibm.com
IBM Research
New Delhi, India

Yogish Sabharwal
ysabharwal@in.ibm.com
IBM Research
New Delhi, India

Srikanta Bedathur
srikanta@cse.iitd.ac.in
Indian Institute of Technology
New Delhi, India

Abstract

Large language models (LLMs) offer a promising, faster and cost-effective approach to code translation. Existing research work on code translation using LLMs assumes improving functional accuracy of the translated code as the primary objective. In this paper, we argue that while it is crucial that the translated code be functionally accurate, it is not the only important factor in code translation, particularly when the translation task involves multiple interdependent source files rather than isolated programs or code snippets. Maintaining a broad *structural* equivalence with the original source code is desirable for higher quality, maintainable, compatible, and easily integrable codes. We present *TranSCoT*, a novel and easy-to-implement technique, that leverages a chain-of-thought reasoning to create systematic prompts that steer LLMs through the code translation process. TranSCoT not only enhances functional accuracy by reducing numerous compilation errors but also ensures that the translated code maintains a broad structural resemblance to the original source, thereby improving maintainability and understandability. Additionally, we introduce a novel *quality metric* to assess code translation quality by evaluating structural similarities between the source and the translated code. We empirically demonstrate the efficacy of TranSCoT prompting approach through evaluations over several benchmarks. For various open-source LLMs, TranSCoT prompts yield translations of significantly higher quality, both structurally and functionally, compared to the existing state-of-the-art prompting techniques.

*Both authors contributed equally to this research.

Conference acronym 'XX, XXX-X-XXXX-XXXX-X/XXXX/XX

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/XXXXXXX.XXXXXXX

Keywords

Code Translation, LLM, Structured Chain-of-Thought, TranSCoT, Translation Quality, Quality Metric, Quality Score

ACM Reference Format:
Lalit Meena, Shashank Govindappa, Monika Gupta, Anamitra Choudhury, Vijay Arya, Yogish Sabharwal, and Srikanta Bedathur. 2025. TranSCoT: Towards Improving Quality of Code Translations. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Enterprises and developers frequently encounter the need to migrate legacy codebases to modern languages for performance enhancement and maintainability. LLMs offer a potentially faster, less expensive, and scalable solution for automating code translation. However, several challenges arise when using LLMs for code translation. Pan *et al.* [17] conducted a large-scale evaluation and taxonomy of translation accuracy bugs that stem from the syntactic and semantic differences between the source and target languages, which manifest as compilation, runtime, or test case failure errors in the translated code. To ensure strong functional equivalence between the source and the translated programs, researchers have explored the use of methods such as reinforcement learning, feedback from compilers, and symbolic equivalence checkers [11, 17, 21] etc. While functional equivalence is necessary for successful code translation, it is not the sole factor to consider when translating codebases that span multiple interdependent files, with a view toward their long-term maintenance.

Ibrahimzada *et al.* [10] highlighted that translation techniques that work well on standalone programs can fail to generalize to the scale and complexity of real-world projects, which have inter- and intra-class dependencies. Their work outlined several challenges in the process of repository-level code translation. In this work, we present yet another challenge in the process of interdependent code translation.

Consider the code translation example in Figure 1, Example 1(a). The original Java code defines two nested classes: *HackingABCEdit* and *LightScanner*. The static member class *HackingABCEdit* defines

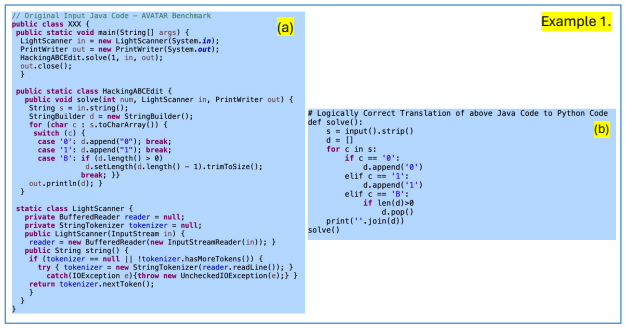


Figure 1: Example 1(a) An input Java source code, (b) Python translation (using *GPT-4*) of the given Java code.

a *solve* method, which reads an input string and uses it to iteratively build and print an output string. The other static member class, *LightScanner*, has methods to initialize an input stream and read an input string from it. The main class *XXX* defines a *main* method that serves as a test case for the *solve* method of the nested class *HackingABCEdit*.

Next, consider its Python translation (Figure 1, Example 1(b)) generated by *GPT-4*. The translation is syntactically and logically correct, but all class and subclass structures from the original Java code have been removed. The basic functionality of the program, *viz*., to read an input string and use it to generate and print an output string, has been placed into a global method. This translation, although correct, may not be appropriate in many settings. What if the original Java program (Figure 1, Example 1(a)) is part of a larger Java package that needs to be fully translated to Python on a file-by-file basis? If, in the package, there are other classes that parameterize and instantiate the *HackingABDEdit* object and call the *solve* method, then the inter-relationships between these classes could break after translation.

As noted in the above example, maintaining a broad *structural* resemblance (in terms of the declared public classes, global variables, methods, and their corresponding call graphs, etc.) with the original code is also critically important to ensure understandability, compatibility, integrability, and ease of maintenance of the translated code.

In this work, we address a critical research question: *How can we effectively translate programs from one programming language to another while maintaining a broad structural similarity between the source and translated programs?* Based on this, we make the following key contributions:

- We propose a novel, lightweight approach called ***TranSCoT***, designed to guide LLMs in effectively translating programs such that the translated code is broadly structurally aligned with the original code and also improves functional accuracy as well.
- Based on structural mismatch issues between the input source code and the translated output code, we introduce a new category of translation issues referred to as ***Translation Quality Bugs***. We subsequently present an innovative quality metric, ***QualScore***, to evaluate the quality of a code translation by assessing the structural similarities between the source codes and the translated codes.

2 Related Work

Transpilers like java2python [15], py2java [7], TSS code converter [2], TransCoder-ST [19] are manually-crafted custom programs that leverage traditional compiler techniques, such as parsing and ASTs, to translate code from a specific language to another. However, many transpilers have a disclaimer stating that the translated code may not readily compile and run.

Xie et al. [24] investigate data augmentation techniques by enhancing it with various reference translations to increase the variability of target translations. Jana et al. [11] integrate reinforcement learning into the LLM fine-tuning process. Similarly, PPOCoder [21] is an RL-based code translation framework that utilizes CodeBLEU-inspired feedback in the LLM training process. Wang et al. [22] proposed a repository-level code translation benchmark and conducted a detailed error analysis highlighting the current LLMs' deficiencies in repository-level code translation. Latest research work [9] on the automation of repository-level code translation suggests techniques to effectively translate both source code and test code to ensure that the translation preserves the functionality of the source program. All these work focus on the functional accuracy rate; however, their impact on translation quality, as discussed in our work, is not known.

Chain-of-Thought (CoT) prompting has emerged as a groundbreaking tool in Natural Language Processing for its efficacy in complex reasoning tasks. The key paradigms of CoT-prompting include Zero-Shot-CoT [12], which uses stepwise reasoning prompts to guide models without prior training on specific examples; Manual-CoT [23], which employs manually crafted demonstrations to teach models step-by-step reasoning; and Auto-CoT [26], which automatically constructs reasoning prompts by identifying patterns in data, reducing the need for human intervention. While Chain-of-Thought (CoT) has not been directly applied to code translation tasks, it has been explored in related areas such as code generation. Notable examples include CodeCoT [8], Structured CoTs (SCoTs) [13], and COTTON [25]. SCoTs, in particular, utilize the structural information in source code during intermediate reasoning steps to produce well-structured code. Our approach, *TranSCoT*, applies CoT with structure-information-based reasoning to code translation to enhance both translation accuracy and quality.

3 Challenges in Code Translation

To study the various challenges involved in automated code translation, we analyzed Java to Python (and vise-versa) code translations generated by several LLMs on 500 standalone programs (250 Java and 250 Python) from the *AVATAR* benchmark [4]. Three open-source LLM: *StarCoder* [14], *Code Llama* [20], *Granite-8B-Code-Instruct* [3] , and a closed-source LLM : *GPT-4* [1] were used to generate these code translations. A basic *vanilla* prompt [17] was used during these translations.

We chose the language pair (*Java*, *Python*) for our study and experiments, as these represent two different programming paradigms.

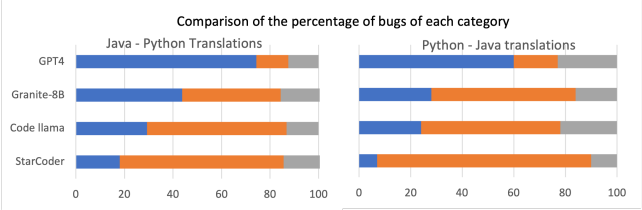


Figure 2: Comparison of percentage of bugs of each category in translated Java and Python codes. Blue - % of Success, Orange - % of Compilation/Runtime Bugs, Gray - % of Test Case Failure. A translation is considered successful if the translated code compiles, passes runtime checks, and all existing test cases.

The fact that Java is strongly typed and object-oriented, while Python is a loosely typed, multi-paradigm language presents various challenges for LLMs ([11], [17]) during code translations.

3.1 Translation Accuracy Bugs

The foremost well-defined and well-understood challenge in code translation through LLMs is the introduction of a large number of translation bugs in the output code [17], [22]. The differences in programming paradigms, syntactic and semantic distinctions between the two languages, make the automatic translation of code from Java to Python (and vise versa) a daunting task. Figure 2 presents a comparison of the percentage of bugs in various categories in the translated Python and Java code, as introduced by the different translating LLMs.

Python programmers often structure solutions in the form of short functions that are well-suited for multi-threading applications. This functional programming paradigm differs from Java's object-oriented paradigm and results in many compilation and run-time errors (Figure 2) in the translated Java code. In Java code for open-source LLMs, between 53% - 82% of bugs are compilation or runtime failure bugs. In Python code, for open-source LLMs, between 40% - 67% of bugs are compilation or runtime failure bugs.

During our experiments, we observed that if, instead of a basic *vanilla* translation prompt, we instruct the LLMs to closely follow a well-structured, progressive, chain-of-thought based reasoning approach to translate code, it helps reduce incomplete or syntactically incorrect translations, thereby considerably reducing a significant number of compile-time errors.

3.2 Translation Quality Bugs

We now introduce a new challenge in code translation - *Translation Quality Bugs*. A translation is generally considered successful if it compiles well, passes all runtime checks, and the existing test cases pass on the translated code. However, we argue that this is not sufficient to generate high-quality, maintainable, and integrable code. We presented an example related to this in the introduction section, Figure 1.

Let us take another example of a Python program (Figure 3, Example 2(a)). An integer variable named a, and a method named $func$, which computes and prints the sum of the digits of the integer variable a, are defined in the global scope of this program. Next,

Example 2.

```python
// Python Source Code - CodeNet Benchmark
a = int(input())
def func(a):
    digits = []
    temp = a
    while temp != 0:
        digits.append(temp%10)
        temp = int(temp/10)
    result = sum(digits)
    if result == 1:
        print('10')
    else:
        print(result)                    (a)
func(a)
```

```java
// Logically Incorrect Java Translation of Python Code
public class YYY{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int a = sc.nextInt();
        int sum = 0;
        while(a!= 0) {
            sum += a % 10;
            a /= 10; }
        System.out.println(sum);          (b)
}}
```

Figure 3: Example 2(a) An input Python source code, (b) Java translation (using *StarCoder*) of the given Python code.

consider its Java translation (Figure 3, Example 2(b)) generated by *StarCoder*. The translation is logically incorrect due to the removal of the last *if-else statement block* in method $func$. Apart from this, another critical point to note in the translated Java code is that all method declarations and global variable declarations have been discarded while retaining only the basic functionality of the program in a monolithic *main* method. This raises the same concern as in the previous example. The original Python program (Figure 3, Example 2(a)) may be part of a larger Python package that needs to be fully translated to Java. Therefore, after translation, the connectivity between programs that call the *func* method and/or access the global variable a would completely break.

Code translations like those presented in Figure 3 and Figure 1 showcase *Translation Quality Bugs*, which can only be addressed by preserving the structure of the original program (to the extent allowed by the target language) after translation. In the next section, we present LLM-based code translation techniques aimed at achieving this.

4 METHODOLOGY

In this section, we illustrate the various approaches we experimented with to improve both the accuracy and the quality of code translation. Building on our insights, these approaches aim to enhance LLMs' comprehension of the translation task by offering pertinent information and detailed descriptions of the translation task with illustrative examples, without the need for any extended model pretraining or finetuning. All the suggested approaches work with prompts and are, therefore, lightweight, low-cost, and easily generalizable across multiple models/datasets. The experimental results of these approaches are summarized in Tables 3 and 1. Detailed prompt templates for each of these approaches are provided in the supplementary material.

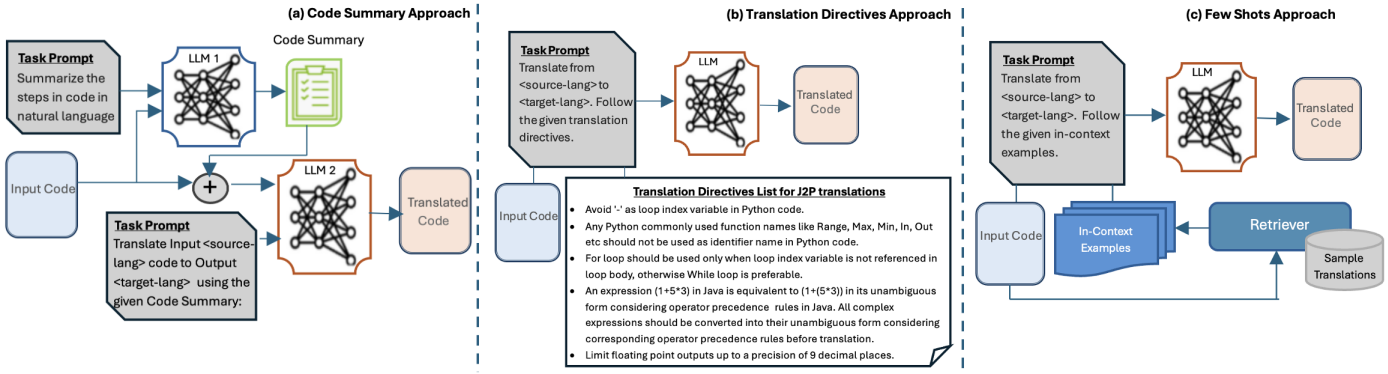


Figure 4: Overview of (a) Code Summary approach, (b) Translation Directives Based Approach, (c) Few Shots Based Approach.

4.1 Vanilla approach - Baseline

It is a simple directive to translate code from a given programming language to another and serves as the baseline approach.

4.2 Code Summary based Approach

A basic vanilla instruction can be enhanced by providing additional relevant information, such as natural language code summaries or pseudocode, sample input-output pairs, and program analysis data, such as type information or abstract syntax trees (ASTs). This additional context can improve LLM's understanding of the source code, resulting in more accurate translations. One of these experiments is conducted by providing LLMs with natural language code summaries along with the input source code (see Figure 4(a)). For this approach, we experimented with Java-to-Python (J2P) and Python-to-Java (P2J) translations using *GPT-4* to generate the code summaries and *StarCoder* to translate the code. We observed that, even with well-crafted code summaries, there was only a modest improvement in the model's overall translation accuracy. Upon manual checking of the translated code files, we observed no significant impact of this approach on translation quality. Detailed results of the experiment are presented in Table 3 and discussed in Section 6.

4.3 Translation Directives based Approach

Test case failed bugs often stem from logical errors in the translated code, which tend to be more language-specific. To address these logical translation errors and improve translation accuracy, we explored the effectiveness of incorporating a *Translation Directives List* in the translation prompt (see Figure 4(b)). This list was developed by manually analyzing and generalizing the common causes of logical translation errors in the test case failure category of translation bugs in J2P translations. We observed that although this technique is effective, it is very language-specific. Results of experiments with this approach are presented in Table 3.

4.4 Few Shots based Approach

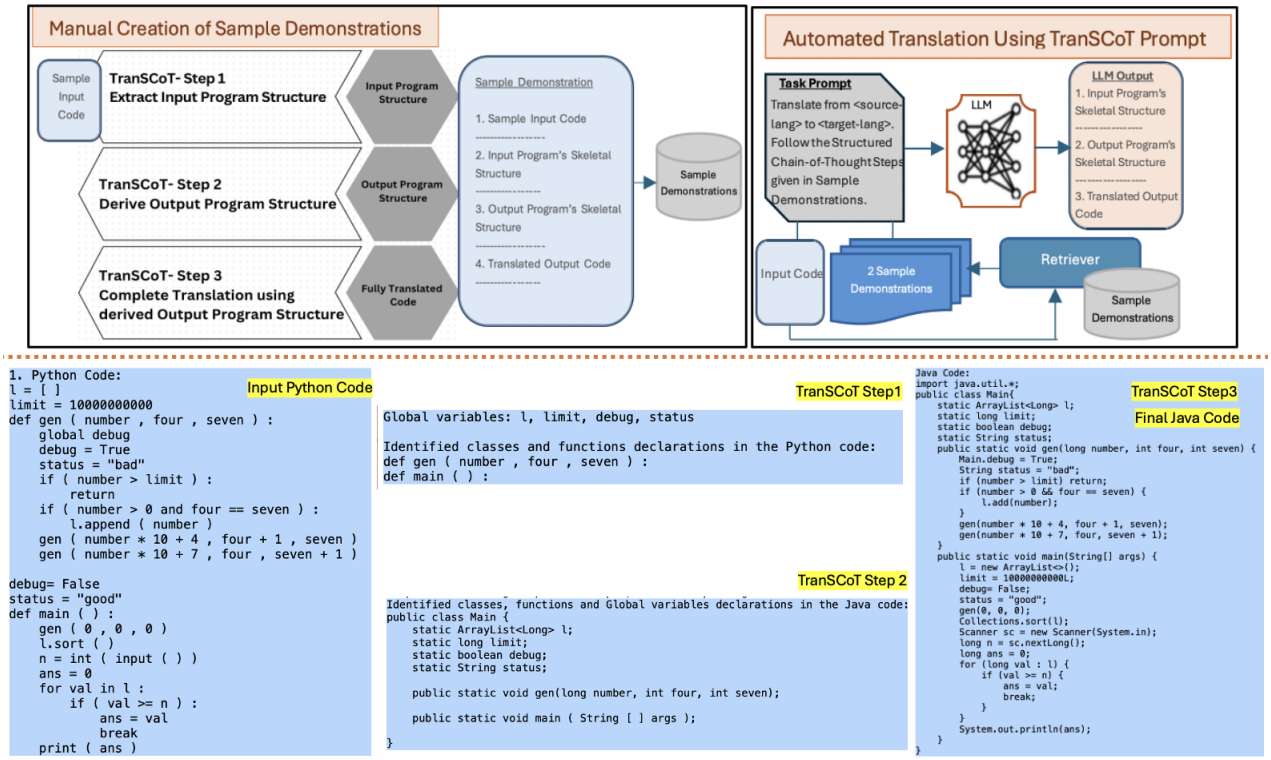
Few-shot prompting [5, 23] facilitates in-context learning (ICL) by providing the model with a few examples to illustrate the task and enhance its understanding. For the few-shots based approach (see Figure 4(c)), we first manually crafted a small pool of 5 samples, accurate, and high-quality translations from the data set. These samples were crafted to represent a range of structural variations, including different input/output formats, nested classes, and global methods and variables. A simple retriever module would randomly select three sample translations (only three, to keep token length issues in check) to serve as in-context examples for the model. We experimented with J2P and P2J translations using this approach on *StarCoder* and observed that this approach results in an impressive increase in the model's translation accuracy. Detailed results of this experiment are presented in Table 3.

4.5 TranSCoT: Structured Translation with CoT

Few-shot prompting combined with Chain-of-Thought (CoT) reasoning [12] is a highly effective technique in NLP for tackling complex tasks. It involves providing demonstrations that consist of a question, a chain of intermediate reasoning steps, and the expected answer. Given that accurate and effective source code translation requires LLMs to reason not only about the program logic but also about its structure, our *TranSCoT* approach integrates structure-consistent CoT reasoning and few-shot learning. This combination allows us to craft elaborate yet systematic prompts that guide LLMs through the code translation process with great precision. The steps involved in the creation of a TranSCoT prompt are outlined below (see Figure 5).

4.5.1 Creation of Sample Demonstrations Pool. A carefully curated set of representative translation demonstrations is first created by manually following the structured CoT reasoning steps (Figure 5) as described in the next subsection. It is crucial that these demonstrations vary based on factors such as logical and structural complexity. When translating codebases from diverse platforms, including samples from each platform is advantageous. For our experiments and evaluations, we assembled a pool of 16 different translation demonstrations, two from each platform, as listed in Figure 6.

4.5.2 Structured-Chain-of-Thought Reasoning Steps. ***Step 1. Extract the Skeletal Structure of the Input Program:*** The first step in the reasoning chain is to identify the skeletal structure of the input source code. The term skeletal structure, as used in our




```
1. Python Code:
l = [ ]
limit = 10000000000
def gen ( number , four , seven ) :
    global debug
    debug = True
    status = "bad"
    if ( number > limit ) :
        return
    if ( number > 0 and four == seven ) :
        l.append ( number )
    gen ( number * 10 + 4 , four + 1 , seven )
    gen ( number * 10 + 7 , four , seven + 1 )

debug= False
status = "good"
def main ( ) :
    gen ( 0 , 0 , 0 )
    l.sort ( )
    n = int ( input ( ) )
    ans = 0
    for val in l :
        if ( val >= n ) :
            ans = val
            break
    print ( ans )
```

```
Global variables: l, limit, debug, status

Identified classes and functions declarations in the Python code:
def gen ( number , four , seven ) :
def main ( ) :
```

```
Identified classes, functions and Global variables declarations in the Java code:
public class Main {
    static ArrayList<Long> l;
    static long limit;
    static boolean debug;
    static String status;

    public static void gen(long number, int four, int seven);

    public static void main ( String [ ] args );
}
```

```
Java Code:
import java.util.*;
public class Main{
    static ArrayList<Long> l;
    static long limit;
    static boolean debug;
    static String status;
    public static void gen(long number, int four, int seven) {
        Main.debug = True;
        String status = "bad";
        if (number > limit) return;
        if (number > 0 && four == seven) {
            l.add(number);
        }
        gen(number * 10 + 4, four + 1, seven);
        gen(number * 10 + 7, four, seven + 1);
    }
    public static void main(String[] args) {
        l = new ArrayList<>();
        limit = 10000000000L;
        debug= False;
        status = "good";
        gen(0, 0, 0);
        Collections.sort(l);
        Scanner sc = new Scanner(System.in);
        long n = sc.nextLong();
        long ans = 0;
        for (long val : l) {
            if (val >= n) {
                ans = val;
                break;
            }
        }
        System.out.println(ans);
    }
}
```

Figure 5: Overview of the TranSCoT approach. A sample P2J translation demonstration is shown.

approach, refers to the high-level structured representation of code. It includes declared code components, such as classes, methods, and global variables, required to complete the logical flow and functionality.

Step 2. Derive the Skeletal Structure of the Output Program: With the extracted skeletal structure of the input code, the subsequent step is to determine an equivalent skeletal structure for the target code. This step ensures that the skeletal structure of the source code is broadly maintained during translation. The acceptable parallels are based on the shared object-oriented programming principles, such as class definitions, inheritance, and method declarations. We assume that these commonalities provide a sufficient framework to maintain the integrity of the skeletal structure across the two languages.

Step 3. Final Translation Step: The final step involves completing the translation by filling in the derived skeletal structure of the output program. The LLM is instructed to account for library imports and input-output formatting, ensuring that the generated code is syntactically and semantically correct.

A manually-crafted sample Python to Java translation demonstration using these steps is shown in Figure 5. The 3-step reasoning process of TranSCoT aims to direct LLMs to first understand a given input program's structure, then create a broad outline structure for the output program, and subsequently fill in this structure. This gives us two benefits - (1) there are fewer chances for LLMs to hallucinate or generate structurally or syntactically incomplete translations, thereby reducing a lot of compilation and runtime errors in the translated code, and (2) the translated code broadly structurally resembles the given input code, and this significantly improves quality, maintainability, integrability, and understandability of the translated code.

4.5.3 Automated Translation Using TranSCoT. For any given code translation task, our approach begins with a retriever module selecting two similar translation demonstrations to use in the TranSCoT prompt, ensuring that the prompt size always remains manageable. In our experiments, the selection of demonstrations is based on the platform similarity between the input sample and the demonstration samples; however, structural similarity can also be used as a criterion. Once the appropriate demonstrations are selected, a TranSCoT prompt is created. It begins with manually crafted expected outputs of the aforementioned steps for the two selected in-context demonstrations. For the given translation task, LLMs are expected to generate all the intermediate and final step's output along the lines of the two in-context demonstrations. Detailed results of all experiments on this approach are included in Table 1.

5 Experimental Setup

This section provides a detailed account of the experimental setup, including the benchmarks and code samples used in the experiments, the LLMs used, and the evaluation metrics.

LLMs: The following LLMs were used in our experiments: *StarCoder*, *Code Llama*, *Granite-8B-Code-Instruct*, *DeepSeek-Coder-V2-Lite-Instruct* , and *GPT-4*. For all experiments, with all LLMs, the

	No. of unique code files		
	platform	Java	Python
AVATAR	atcoder	127	127
	codeforces	123	123
	# Total	250	250
CODENET	# Total	200	200
AVATAR_TC	aizu	190	190
	atcoder	97	97
	codeforces	401	401
	codejam	4	4
	geeksforgeeks	995	995
	leetcode	18	18
	projecteuler	41	41
	# Total	1746	1746

Figure 6: Benchmark Statistics.

context length limit was set to 4096, which was sufficient for the size of the programs in the evaluated benchmarks. All experiments were conducted on a single A100 Nvidia GPU.

Benchmark Datasets: The benchmark datasets for our experiments are presented in Figure 6. The **AVATAR** (jAVA-pyThon prograAm tRanslation) dataset [4] contains multiple test cases for 250 Java and Python examples, while the **CODENET** dataset [18] includes 200 Java-Python pairs, enabling an exhaustive functional accuracy evaluation of the translation models. **AVATAR_TC** dataset [11] comprises large-scale Java and Python codes derived from multiple programming platforms. For comparative analysis with state-of-the-art techniques, we used a subset of 1,746 Java-Python translation pairs, consistent with the subset used by Jana et. al. [11]. For all experiments, we anonymize the input source code to prevent any code-bias in case the translating LLM has already seen the code during its initial training.

5.1 Translation Accuracy Metrics

As in previous studies [17], [11], we evaluate the accuracy of translated code using two key metrics: ***Functional Accuracy (Func.Acc.)*** - it measures the percentage of translated programs that pass all their test cases out of a set of input benchmark programs, and, ***Compilation Accuracy (Comp.Acc.)*** - it reflects the percentage of translated programs that successfully compile in the set. We report *Pass@1* and *Pass@5* [6] numbers. For Pass@1, we employed the greedy decoding algorithm that selects only the top translation with the highest log probability. For Pass@5, we sampled 10 translations per run and used a temperature setting of 0.2 to obtain more deterministic and task-relevant generations.

5.2 Translation Quality Metrics

Given that our *TranSCoT* method relies on mapping structural components from the source code to equivalent structures in the target language, it is crucial to assess the quality of the final translation to ensure the efficacy of the reasoning process. For this purpose, we use the following two metrics: first, we use **LLM as Judge**, where we prompt an LLM to evaluate the structural similarity between the original and translated code in terms of the declared methods and function calls, and generate a score between 0 and 1 (0 being not similar at all, and 1 being total similarity). Second, we introduce a novel *Structural Quality Assessment Score* (***QualScore***) that further emphasizes the structural closeness by directly measuring the similarity between the *Structural Call Graphs* of the input code and the translated output program based on the *Normalized Graph Edit Distance* between them.

5.2.1 Structural Call Graph: Methods within a program are fundamental structural components. To capture their relationships, we define a structural call graph, $\mathbb{G}_{pgm}$, as an enhanced version of a control-flow graph that incorporates the calling relationships between methods. Unlike a standard control-flow graph, $\mathbb{G}_{pgm}$ includes a node for every method in the program, even if a method is not called by any other method.

$$\mathbb{G}_{pgm} = (\mathbb{V}_{pgm}, \mathbb{E}_{pgm})$$
$$\mathbb{V}_{pgm} = \{v : v \; is \; a \; method \; in \; the \; program\}$$
$$\mathbb{E}_{pgm} = \{(v_i, v_j) : v_i, v_j \in \mathbb{V}_{pgm}, \; and, \; v_i \; calls \; method \; v_j\}$$

Structural call graphs are based on methods and the ways in which they call each other. Since methods are an interface through which programs interconnect and interact, it is essential that these method boundaries and method calls be preserved between the input code and the translated output code.

5.2.2 Normalized Graph Edit Distance. We adopt a normalized graph edit distance to measure the similarity between structural call graphs. Formally, the graph edit distance between two graphs, g_1 and g_2, $GED(g_1, g_2)$, is defined as:

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

where $\mathcal{P}(g_1, g_2)$ denotes the set of edit operations that transform g_1 into a graph isomorphic to g_2, and $c(e) \geq 0$ represents the cost of each graph edit operation e. For simplicity, we assign a cost of 1 to each operation in all our experiments.

Let ϕ be a null graph with zero nodes and edges. The *Normalized Graph Edit Distance* (0..1) is expressed as:

$$normalized\_GED(g_1, g_2) = \frac{GED(g_1, g_2)}{GED(g_1, \phi) + GED(\phi, g_2)}$$

5.2.3 Defining QualScore. For a given translation, the greater the normalized graph edit distance, the greater the deviation of the translated code's structure from the source code's structure, and the poorer its quality. We compute the QualScore of a given translation as :

$$QualScore = 1 - normalized\_GED(g_1, g_2)$$

For a given set of input benchmark programs and their corresponding translated output programs, the QualScore is the average of the QualScores of all translations in the set.

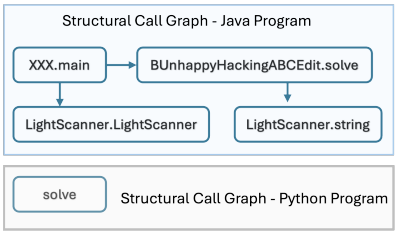


Figure 7: Structural Graphs for Java, Python programs of Figure 1.

Table 1: Translation results comparison of TranSCoT with various baselines for J2P and P2J translations.

Benchmark	LLMs	Approach	Java-Python Translations				Python-Java Translations			
			Accuracy			Trans. Quality	Accuracy			Trans. Quality
			Pass@1		Pass@5		Pass@1		Pass@5	
			Func.Acc.	Comp.Acc.	Func. Acc.	QualScore	Func. Acc.	Comp. Acc.	Func. Acc.	QualScore
AVATAR	StarCoder	Vanilla	18.1	67.9	27.5	0.04	6.4	34	9	0.15
		TranSCoT	45.4	90.8	53.2	0.7	28.4	59.2	35.9	**0.59**
	Code Llama	Vanilla	29.2	86	45.5	0.01	22.4	62.8	33	0.52
		TranSCoT	38.4	90	45.1	**0.82**	26.8	68	34.6	0.55
	Granite8b	Vanilla	43.8	93.2	52.4	0.01	28.4	66.4	35.6	0.57
		TranSCoT	44.6	96	57.9	0.64	32.4	67.6	38.2	**0.59**
	Deepseek	Vanilla	57.6	92.4	64.32	0.08	51.3	79.1	62.6	0.54
		TranSCoT	**86.2**	**95.3**	**89.5**	0.67	**64.7**	**84.6**	**68.9**	0.55
	GPT	Vanilla	75.2	–	–	0.1	–	–	–	–
		TranSCoT	69.4	–	–	0.7	–	–	–	–
CODENET	StarCoder	Vanilla	29.5	68.5	37.2	0.02	9	18.5	37.2	0.09
		TranSCoT	52	88	57.9	**0.64**	53.5	68.5	57.9	0.67
	Code Llama	Vanilla	48.5	90.5	57.4	0.01	42	58	56.1	0.54
		TranSCoT	49.2	91	56.1	**0.64**	62.5	78.5	57.4	**0.79**
	Granite8b	Vanilla	49	89.5	55.6	0.01	56.49	68.5	65.6	0.65
		TranSCoT	55	89	64.3	**0.64**	66.5	76.5	68.3	0.75
	Deepseek	Vanilla	57.32	85.5	64.3	0.44	63.7	73.4	68.3	0.66
		TranSCoT	**83.15**	**94.3**	**86.1**	0.57	**76.5**	**90.2**	**79.1**	0.77

Table 2: Translation results comparison of TranSCoT with CoTran baseline for J2P and P2J translations.

		AVATAR_TC													
		aizu		atcoder		codeforces		leetcode		geekforgeeks		projecteuler		overall	
		CoTran	TranSCoT	CoTran	TranSCoT	CoTran	TranSCoT	CoTran	TranSCoT	CoTran	TranSCoT	CoTran	TranSCoT	CoTran	TranSCoT
Java-Python	**Func.Acc.**	14.2	32.6	25.8	51.54	18	45.38	55.6	50	74.6	67.33	12.2	31.7	53.89	56.47
	QualScore	0.06	0.72	0.04	0.88	0.02	0.76	0	0.97	0.2	0.95	0.15	0.65	0.13	0.88
Python-Java	**Func.Acc.**	14.73	32.1	30.92	54.63	24.43	44.88	24.77	38.88	68.64	69.84	12.19	19.51	48.68	57.76
	QualScore	0.44	0.51	0.68	0.65	0.72	0.63	0.15	0.2	0.79	0.85	0.09	0.36	0.7	0.73

Example: Figure 7 displays the structural call graphs for the Java and Python programs shown in Figure 1, Example 1. It is clear that the structural call graphs of these two programs do not align well and a minimum of six elementary operations are required to transform the Python call graph, resulting in a graph edit distance of 6 for this pair. The normalized_GED value is 6/8 = 0.75, and QualScore = 0.25. Although these Java and Python programs are functionally equivalent, the structure of function calls and definitions in the Java code is not preserved in the generated Python code, resulting in a translation that, while functionally correct, exhibits poor quality.

6 Results Analysis

Experiment 1: On the AVATAR benchmark, for J2P and P2J translations using the StarCoder LLM, we first compare the functional accuracies (refer to table 3) achieved by various alternative prompting approaches outlined in Section 4. The TranSCoT approach clearly achieves a much higher functional accuracy and outperforms all of these approaches.

Code Summary Approach - We observed that the pre-translation step of generating precise natural language code summaries presents

Table 3: Comparison of various approaches with vanilla baseline for J2P and P2J translations.

250 Java Code Samples - AVATAR Benchmark J2P Translation, Pass@1, Translating LLM - StarCoder		
Approach	**Func.Acc.**	**Comp.Acc.**
Vanilla	18.1	67.9
Code Summary	22.08	70.42
Translation Directives	21.18	68.3
Few Shots	30.12	74.88
TranSCoT	45.4	90.8

250 Python Code Samples - AVATAR Benchmark P2J Translation, Pass@1, Translating LLM - StarCoder		
Approach	**Func.Acc.**	**Comp.Acc.**
Vanilla	6.4	34
Code Summary	10.13	42.1
Few Shots	18.34	47
TranSCoT	28.4	59.2

a significant cost and time challenge. Additionally, even with well-crafted code summaries, there was only a modest improvement of approximately 2-4% in successful translations, indicating that the LLMs were not able to extract much advantage from the code summaries, probably because these were disintegrated from the corresponding code.

Translation Directives Approach - The main limitation of this approach is that it is very language-specific, and any Translation Directives list cannot be considered absolutely complete; it has to grow as more erroneously translated code is examined. Also, it does not ensure that LLMs will fully comprehend or adhere to these directives. Nonetheless, it does help in partially resolving some logical errors, as is clear from the increase in J2P functional accuracy scores compared to the vanilla approach.

Few-shot Prompting Approach - The few-shot prompting approach resulted in an impressive increase in translation accuracy for all translations, indicating that providing in-context examples helps LLMs to better understand the tasks to be done. However, to avoid any unwanted bias, it is crucial to select these examples with care, considering factors such as diversity and the order of examples [16].

Experiment 2: We next perform an extensive comparison of the functional accuracy, compilation accuracy, and quality scores achieved by TranSCoT with those achieved by the vanilla prompt for both J2P and P2P translations on the AVATAR and CODENET benchmarks. Table 1 presents these comparison results. The TranSCoT prompt method demonstrates significant improvements across almost all models, outperforming the vanilla prompt technique in both functional accuracy and compilation accuracy. StarCoder and DeepSeek, in particular, perform exceptionally well, achieving at least a 22% improvement in functional accuracy. For example, J2P translations on the AVATAR dataset using the vanilla prompting technique yield only 18% accuracy on StarCoder, whereas using TranSCoT boosts it to 45.4%, resulting in a gain of 27.3%. We also observe notable improvements in P2J translations for the CODENET dataset, with Code Llama achieving 62.5% compared to 42% with the vanilla prompting technique. The reason for this overall improvement using the TranSCoT method is the substantial increase in the compilation accuracy rate across all models for both J2P and P2P translations. In terms of quality score as well, for J2P translations, the vanilla approach performs poorly, with an average value of 0.12. In contrast, TranSCoT achieves a significantly better score of 0.65 for J2P translations on both benchmarks.

We tested J2P translations using the AVATAR benchmark on GPT-4, comparing vanilla prompting with TranSCoT. Although slightly better functional accuracy was achieved using basic vanilla prompting, it definitely produced much lower-quality translations, as indicated by a poor QualScore of 0.1 compared to the 0.7 achieved by TranSCoT.

Table 4 presents the quality comparison of various translation sets with *LLM as a Judge* Quality Metric. GPT-4o was used as the judge LLM. These results revalidate the efficacy and supremacy of the TranSCoT approach in generating high-quality translations.

Experiment 3: The *CoTran* tool [11] is a state-of-the-art tool that integrates reinforcement learning into the LLM fine-tuning process, incorporating compiler feedback and symbolic execution-based equivalence testing to ensure functional equivalence between input and output programs. To compare with CoTran, we applied our approach to translate programs from 8 different programming platforms in the AVATAR_TC benchmark using the Granite8B LLM. We then compared the functional accuracies achieved by TransCoT with those claimed by the CoTran tool on the same test set. Table 2 presents the results. Our tool achieves an overall functional accuracy of 56.47% for J2P translations, surpassing the best result of 53.89% reported by the CoTran tool. For P2J conversions, our approach shows an even greater improvement of 57.76% compared to 48.68% reported by the CoTran tool. CoTran tool is claimed [11] to outperform 14 other state-of-the-art tools for J2P and P2J translations. By outperforming CoTran, we demonstrate the efficacy of an easy-to-implement prompting approach like TransCoT against these 14 other baselines and other, not-so-easy-to-implement LLM fine-tuning techniques.

Table 4: Quality Comparison of TranSCoT approach with vanilla baseline for J2P and P2J translations.

250 Java Code Samples - AVATAR Benchmark J2P Translation		
LLMs	**Approach**	**LLM as a Judge Score (Avg, Std. Dev)**
StarCoder	Vanilla	(0.4, 0.27)
	TranSCoT	(0.73, 0.14)
CodeLlama	Vanilla	(0.43, 0.29)
	TranSCoT	(0.7, 0.12)
Granite8b	Vanilla	(0.51, 0.27)
	TranSCoT	(0.78, 0.14)
DeepSeek	Vanilla	(0.52, 0.28)
	TranSCoT	(0.75, 0.11)

250 Python Code Samples - AVATAR Benchmark P2J Translation		
LLMs	**Approach**	**LLM as a Judge Score (Avg, Std. Dev)**
StarCoder	Vanilla	(0.51, 0.23)
	TranSCoT	(0.69, 0.12)
CodeLlama	Vanilla	(0.57, 0.21)
	TranSCoT	(0.59, 0.11)
Granite8b	Vanilla	(0.55, 0.23)
	TranSCoT	(0.6, 0.1)
DeepSeek	Vanilla	(0.61, 0.2)
	TranSCoT	(0.6, 0.1)

7 Limitations and Threats to Validity

Our work explores J2P/P2J translations for single-file programs derived from competitive programming-platforms. However, we still need to explore how TranSCoT prompting will perform for other language pairs like procedural and functional language pairs,

or real-world projects with more complex code structures and spanning multiple files.

Given the context length constraints of LLMs, the TranSCoT prompt carries significant contextual limitations. This results in a lack of experimentation with more than two demonstration samples, which could further improve the results of our approach.

8 Conclusions

We introduced a novel, lightweight approach that enhances LLMs' ability to navigate the complexities of code translation. Our approach has been rigorously validated, demonstrating its effectiveness. Additionally, we emphasized the crucial role of evaluating translation quality and introduced an innovative quality metric designed to assess the quality of translated code.

9 Acknowledgements

This work was partially supported by an IITD-IBM AIHN collaborative project. Srikanta Bedathur acknowledges the DS Chair Professor of AI fellowship.

References

[1] 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[2] 2023. TSS. The Most Accurate and Reliable Source Code Converters, 2023.(Tangible Software Solutions). https://www.tangiblesoftwaresolutions.com.
[3] 2024. Granite Code Models: A Family of Open Foundation Models for Code Intelligence. arXiv:2405.04324 [cs.AI]
[4] 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).
[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[6] 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. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374
[7] Nikita Fomin. 2019. py2java: Python to Java Language Translator, 2019. https://pypi.org/project/py2java/.
[8] Dong Huang, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024. CodeCoT: Tackling Code Syntax Errors in CoT Reasoning for Code Generation. arXiv:2308.08784 [cs.SE]
[9] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Repository-Level Compositional Code Translation and Validation. arXiv:2410.24117 [cs.SE] https://arxiv.org/abs/2410.24117
[10] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2025. AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation. *Proceedings of the ACM on Software Engineering* 2, FSE (June 2025), 2454–2476. doi:10.1145/3729379
[11] Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh. 2024. CoTran: An LLM-based Code Translator using Reinforcement Learning with Feedback from Compiler and Symbolic Execution. arXiv:2306.06755 [cs.PL]
[12] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.
[13] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599* (2023).
[14] 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).
[15] Troy Melhase, Brian Kearns, Ling Li, Iulius Curt, and Shyam Saladi. 2016. java2python: Simple but Effective Tool to Translate Java Source Code into Python. https://github.com/natural/java2python.
[16] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837* (2022).
[17] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem 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 the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. ACM. doi:10.1145/3597503.3639226
[18] 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)*.
[19] 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.
[20] 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. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]
[21] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based Code Generation using Deep Reinforcement Learning. arXiv preprint arXiv:2301.13816, 2023.
[22] Yanli Wang, Yanlin Wang, Suiquan Wang, Daya Guo, Jiachi Chen, John Grundy, Xilin Liu, Yuchi Ma, Mingzhi Mao, Hongyu Zhang, and Zibin Zheng. 2024. RepoTransBench: A Real-World Benchmark for Repository-Level Code Translation. arXiv:2412.17744 [cs.SE] https://arxiv.org/abs/2412.17744
[23] 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.
[24] Yiqing Xie, Atharva Naik, Daniel Fried, and Carolyn Rose. 2023. Data Augmentation for Code Translation with Comparable Corpora and Multiple References. arXiv:2311.00317 [cs.CL] https://arxiv.org/abs/2311.00317
[25] Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Terry Yue Zhuo, and Taolue Chen. 2023. Chain-of-Thought in Neural Code Generation: From and For Lightweight Language Models. *arXiv preprint arXiv:2312.05562* (2023).
[26] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493* (2022).

10 APPENDIX

10.1 Prompting Template - Code Summary-based Approach

```
// PROMPT1 TEMPLATE
$SOURCE_LANG:
// Unformatted source code
SAMPLE INPUT:
// Sample input
SAMPLE OUTPUT:
// Sample output
Summarize the steps in above $SOURCE_LANG code in natural language by
    considering the sample inputs and outputs
CODE SUMMARY:
// Code summary is generated here
```

```
//PROMPT2 TEMPLATE
Translate below $SOURCE_LANG code to $TARGET_LANG code by considering
    $SOURCE_LANG code and its code summary.
$SOURCE_LANG:
// Unformatted source code
CODE SUMMARY:
// Natural language code summary
$TARGET_LANG:
// Translated code is generated here
```

10.2 Prompting Template - Directives-based Approach

```
Translate below Java code to Python code.
Consider the following rules when translating from Java to Python.
a. Add all required library imports in the Python Code.
b. Make sure the generated code is syntactically correct.
c. Avoid '_' as loop index variable in Python code.
d. Any Python commonly used function names like Range, Max, Min, In, Out etc
    should not be used as identifier name in Python code.
e. For loop should be used only when loop index variable is not referenced in loop
    body, otherwise While loop is preferable.
f. An expression (1+5*3) in Java is equivalent to (1+(5*3)) in its unambiguous form
    considering operator precedence rules in Java. All complex expressions
    should be converted into their unambiguous form considering corresponding
    operator precedence rules before translation.
g. Limit floating point outputs upto a precision of 9 decimal places.
Java:
// Unformatted Java source code
Python:
// Translated Python code is generated here
```

10.3 Prompting Template - Few-Shots-based Approach

```
Translate the $SOURCE_LANG code to $TARGET_LANG code.
1. $SOURCE_LANG:
// Unformatted source code of sample 1
1. $TARGET_LANG:
// Manually translated code of sample 1
----------------
<< Second Sample >>
----------------
<< Third Sample >>
----------------
4. $SOURCE_LANG:
// Unformatted source code to be translated
4. $TARGET_LANG:
// Generated target code
```

Figure 8: General Prompt template P2J_TranSCoT

Demonstration 1.
Python Code:

```
import <<requirements>>
global_var1 = val
# <<more global variable operations>>
def function1():
    # << function definition >>
class HelperClass:
    def __init__(self):
        # << some initializing statements >>
    def helper(self):
        global gvar2
        gvar2 = "new_global"
        # << function definition >>
if __name__=="__main__":
    helper_obj = HelperClass()
    helper_obj.helper()
    print("global variable value 1: ", global_var1)
    print("global variable value 2: ", gvar2)
    function1()
```

Steps: Let's think step by step.
Step 1: Identify global variables in the given code. Global variables are not inside the scope of any function or class declarations.
Global variables: global_var1, gvar2, other global vars
Step 2: Identify classes and functions declarations present in the given code.
Identified classes and functions declarations:

```
def function1():
class HelperClass:
    def __init__(self):
    def helper(self):
if __name__=="__main__":
```

Step 3: Considering Step 1 and Step 2, identify the structure of output Java code.
Identified classes, functions and global variables declarations in the Java code:

```
public class Main {
    static int global_var1;
    static String gvar2;
    static Main.HelperClass helper_obj;
    public static int function1 ();
    static class HelperClass {
        HelperClass ();
        String helper ();
    }
    public static void main (String[] args);
}
```

Step 4: Complete Python to Java translation by considering identified classes, functions and global variables in Step 3.
While generating the Java code, keep track of required library imports to be added in the Java Code.
Make sure the generated code is syntactically correct.
Java Code:

```
import <<requirements>>;
public class Main {
    static int global_var1;
    static String gvar2;
    static Main.HelperClass helper_obj;
    public static int function1() { /* << function definition >> */ }
    static class HelperClass {
        HelperClass () { /* << some initializing statements >> */ }
        String helper () { /* << function definition >> */ }
    }
    public static void main (String [ ] args ){
        global_var1 = val;
        helper_obj = new Main.HelperClass();
        helper_obj.helper();
        System.out.println("global variable 1 value : " + global_var1);
        System.out.println("global variable 2 value : " + gvar2);
        int x = function1();
        // << function definition >>
    }
}
```

****

Demonstration 2 - Second example follows
****

Translation Task
****

Python Code:
source code of Python program to be translated
Steps: Let's think step by step.
LLM will generate from here

Figure 9: General Prompt template J2P_TranSCoT

Demonstration 1.
Java Code:

```
import <<requirements>>
public class Main {
    //<<class variables declarations>>
    public static void main(String[] args)
        { /*<<function definition>>*/}
    static class CustomScannerClass {
        //<<class variables declarations>>
        String next( ) { /*<<function definition>>*/}
        long otherFunctions( ) { /*<<function definition>>*/}
    }
    static class HelperClasses {
        //<<class variables declarations>>
        HelperClasses( ) { /*<<function definition>>*/}
        String helper( ) { /*<<function definition>>*/}
    }
}
```

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, all class and functions declarations should be present in the 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.
Add all required library imports in the generated Python Code.
Make sure your generated code is syntactically correct.
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()
```

****

Demonstration 2 - Second example follows
****

Translation Task
****

Java Code:
source code of Java program to be translated
Steps: Let's think step by step.
LLM will generate from here