Exploring the Impact of the Output Format on the Evaluation of Large Language Models for Code Translation

Marcos Macedo
Queen's University
Kingston, ON, Canada
marcos.macedo@queensu.ca

Yuan Tian
Queen's University
Kingston, ON, Canada
y.tian@queensu.ca

Filipe R. Cogo
Centre for Software Excellence, Huawei Canada
Kingston, ON, Canada
filipe.cogo@gmail.com

Bram Adams
Queen's University
Kingston, ON, Canada
bram.adams@queensu.ca

Abstract

Code translation between programming languages is a long-existing and critical task in software engineering, facilitating the modernization of legacy systems, ensuring cross-platform compatibility, and enhancing software performance. With the recent advances in large language models (LLMs) and their applications to code translation, there is an increasing need for comprehensive evaluation of these models. In this study, we empirically analyze the generated outputs of eleven popular instruct-tuned LLMs with parameters ranging from 1B up to 46.7B on 3,820 translation pairs across five languages, including C, C++, Go, Java, and Python. Our analysis found that between 26.4% and 73.7% of code translations produced by our evaluated LLMs necessitate post-processing, as these translations often include a mix of code, quotes, and text rather than being purely source code. Overlooking the output format of these models can inadvertently lead to underestimation of their actual performance. This is particularly evident when evaluating them with execution-based metrics such as Computational Accuracy (CA). Our results demonstrate that a strategic combination of prompt engineering and regular expression can effectively extract the source code from the model generation output. In particular, our method can help eleven selected models achieve an average Code Extraction Success Rate (CSR) of 92.73%. Our findings shed light on and motivate future research to conduct more reliable benchmarks of LLMs for code translation.

CCS Concepts

• **General and reference** → **Empirical studies**; **Evaluation**; • **Software and its engineering** → General programming languages; • **Computing methodologies** → *Natural language processing*; **Machine translation**;

Keywords

code translation, output format, large language model, LLM, software engineering, benchmarking, evaluation, empirical study, case study

ACM Reference Format:
Marcos Macedo, Yuan Tian, Filipe R. Cogo, and Bram Adams. 2024. Exploring the Impact of the Output Format on the Evaluation of Large Language Models for Code Translation. In *AI Foundation Models and Software Engineering (FORGE '24), April 14, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650105.3652301

FORGE '24, April 14, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0609-7/24/04…$15.00
https://doi.org/10.1145/3650105.3652301

1 Introduction

Automated code translation—translating source code between different programming languages (PLs)—promises to save substantial time and effort for software development teams, while also minimizing the risks associated with manual translation errors and inconsistencies. As more companies seek to migrate their existing software systems from outdated PLs to more contemporary ones, or adapt their software for cloud-based environments where specific PLs are better suited, the value of automated code translation becomes increasingly apparent [44].

Traditionally, code translation is implemented using transpilers [4, 5, 7, 9], tools that use hard-coded rules and program analysis to convert code between languages. These tools are not only costly to develop but also limited to specific language pairs. Furthermore, they often produce translations that do not align with the idiomatic expressions of the target language [41]. Recently, pre-trained models have gained prominence in code translation [23, 37, 38, 41]. These models are pre-trained on large-scale source code datasets and occasionally fine-tuned on code translation pairs. Large language models (LLMs) offer new opportunities in code translation by enabling prompt engineering, an alternative to traditional model parameter adjustments, which leverages *prompts* to guide LLMs in addressing diverse tasks. The introduction of LLMs and prompt engineering has significantly advanced automated software engineering (ASE) tasks, elevating their performance to new records [15].

Interestingly, despite the advancements of LLM-based approaches in different ASE tasks, a recent study by Pan et al. [32] shows that LLMs are yet to prove their reliability to automate code translation, with the rate of translations that can completely

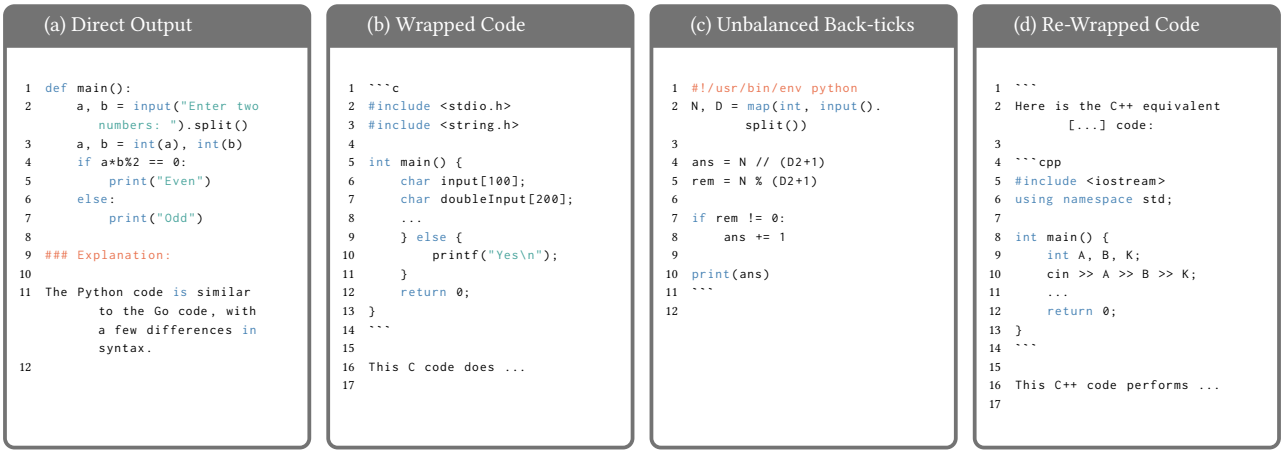

Figure 1: Examples of the observed output formats across the eleven models.

(a) Python code in Direct Output with Additional text (the explanation). **(b)** C code wrapped with three back-ticks followed by the language extension, with Additional text. **(c)** Python code that has three-back ticks at the end but no matching opening back-ticks. **(d)** This output format occurs in RQ2 as the model does not generate code after the back-ticks and instead generates a completely new code block. Some code examples are shortened for brevity.

pass the accompanying unit test(s), i.e., Computational Accuracy (CA) [37], ranging from 2.1% to 47.3% on average. Pan et al. manually categorized the bugs introduced by LLMs in code translation and found that most (77.8%) unsuccessful translations result in compilation errors. In this paper, we postulate that the practical evaluation of LLMs for code translation revolves around the models' capability to generate correctly translated code. Nonetheless, generated translations deemed as invalid during LLM evaluation (e.g. due to compilation errors) often stem from the problem of *inconsistent output format*, not necessarily from the LLMs' limitations for code translation. Inconsistent output format refers to a problem where the translated code is presented in the generated output in a variety of formats and could be interspersed with natural language (ref. Fig. 1). Such inconsistent output formats jeopardize the calculation of many important benchmark metrics for evaluating code translation approaches. More specifically:

- **Text-oriented metrics**: Inconsistent output formats lead to a misestimate of text-oriented evaluation metrics, as generated text other than translated code, such as code explanations, is accounted for in their calculations. These include BLEU [33], CodeBLEU [36], ChRF [34], CrystalBLEU [14] and others. These metrics measure either the overlap of tokens or the semantic similarity between a translation and its reference.
- **Execution-based metrics**: Inconsistent output formats lead to wrong estimations when calculating execution-based metrics such as CA. While the code excerpt in the output can be a potentially valid translation of the reference source code, the whole output fails compilation. This leads to the conclusion that the model was not capable of generating the correct translated code when, in fact, the problem stems from the generated natural language text, which does not comply with the source code syntax.

Therefore, it is important to make researchers and practitioners aware of the impact of inconsistent output format on the evaluations of LLMs for code translation and distinguish between errors arising from the output format itself and those stemming from the quality of the translated code. By doing so, we can more reliably assess the ability of LLMs to translate source code between different PLs.

To validate our hypothesis and explore if the issue of inconsistent output format can be addressed using cost-effective techniques (i.e., prompt engineering and lightweight post-processing), we conduct a case study by empirically investigating the generated output of eleven popular instruct-tuned LLMs (ref. Table 1) for code translation across pairs of different PLs. Our investigation encompassed 3,820 code translation problems, with the source code samples originating from the well-known code translation benchmark CodeNet [35] and the target code samples generated by different LLMs. The dataset covers 20 translation combinations of source-target PL pairs considering five PLs (C, C++, Go, Java, and Python). In particular, we answer the following RQs:

RQ1 ***What are the characteristics of the output formats across LLMs and prompts?*** We analyzed the output formats generated by eleven LLMs and found that these models generate outputs in three distinct source code formats, some of which include natural language (additional text) and some that do not. Of these six possible combinations (output format with or without additional text), only one is directly parsable without further processing.

RQ2 ***To what extent can the output format of LLMs be controlled using prompt engineering and lightweight post-processing?*** We verified that combining prompt instructions to control the output format with a regular expression for code extraction can increase the proportion of source code extracted from the inference output.

RQ3 ***What's the impact of output control on the reported performance of LLMs?*** We use different combinations of prompt and extraction methods and study their impact on the CA metric. We find that the proposed lightweight approach presents an average CA up to 6 times higher than a direct compilation of the LLMs' outputs.

The following are the specific contributions of our paper:

- We show how often LLMs do not consistently output the expected code format for benchmark evaluation in code translation.
- We demonstrate a cost-effective output control method to increase the chances of retrieving the source code from the generation output of the models and calculate its effectiveness in code translation.
- We assess the impact of output control on the reported performance of LLMs in code translation in terms of compilation rate and CA.

To the best of our knowledge, this is the first research work investigating the influence of the output format on LLM-based code translation evaluation. By gaining insights into the expected output formats of LLMs in the context of code translation evaluation, we aim to facilitate progress in this field. Our efforts will raise awareness among researchers and practitioners engaged in benchmark-based evaluations, fostering the pursuit of more dependable and comparable results. Moreover, we have made our replication package publicly available on GitHub[1], encouraging and supporting further research on this topic.

2 Background and Related Work

This section introduces background and related work on automated code translation (Section 2.1), benchmark evaluation (Section 2.2), and prompt engineering (Section 2.3).

2.1 Automated Code Translation

Automated code translation is a long-existing task in software engineering (SE). Existing approaches can be categorized into four classes, rule-based [4, 5, 7, 9], statistical learning-based [21, 29, 30], neural network-based [13], and pre-trained model-based [20, 23, 32, 38, 41]. The first category encompasses tools that utilize program analysis techniques and handcrafted rules to translate code from one programming language (PL) to another. For instance, C2Rust [5] and CxGo [4] are designed to convert C programs into Rust and Go, respectively. Sharpen [9] and Java2CSharp [7] are tools for transforming Java code into C#. While these tools are employed in the industry, their development can be costly. Additionally, they often result in translations that may not align with the idiomatic usage of the target language, leading to challenges in readability for programmers [41]. As representatives for the second category, Nguyen et al. [29, 30], Karaivanov et al. [21] and Aggarwal et al. [12] investigated how well statistical machine translation models for natural languages could apply to code translation. These methods overlook the grammatical structures inherent in PLs. To mitigate this issue, Chen et al. [13] proposed a tree-to-tree neural network that translates a source tree into a target tree by using an attention mechanism to guide the expansion of the decoder. Their approach outperforms other neural translation models designed for translating human languages.

Pre-trained model-based approaches leverage self-supervised learning to train a model from large-scale source code for code translation. Roziere et al. [37] introduced an unsupervised code translation model that only requires source code in multiple PLs without parallel code translation pairs, which are often hard to collect. They developed TransCoder, a model pre-trained on GitHub's open-source projects leveraging three tasks, i.e., masked language modeling, recovering corrected code, and back-translation. Building on this, they later introduced DOBF[23] and TransCoder-ST [38]. DOBF is a model pre-trained to reverse code obfuscation by employing a sequence-to-sequence learning model. TransCoder-ST, on the other hand, utilizes automatic test generation techniques to automatically select and fine-tune high-quality translation pairs for the pre-trained model. Szafraniec et al. proposed TransCoder-IR [41], enhancing TransCoder by incorporating low-level compiler intermediate representations. In addition to models specifically pre-trained for code translation, general-purpose pre-trained models for source code, such as CodeT5 [42] and CodeBERT [17], have also shown promise in code translation tasks. These models can be effectively fine-tuned on parallel code translation pairs, as demonstrated in the work of Jiao et al. [20].

Recently, large language models (LLMs) have gained prominence as highly effective tools for code intelligence tasks, including code translation [15]. Different from previous pre-trained model-based approaches, LLMs are designed to understand and follow human instructions, enabling their straightforward application in various downstream tasks through prompt-based interactions rather than expensive task-specific fine-tuning or pre-training. Pan et al. [32] investigate the effectiveness of LLMs in code translation. The authors collected executable code samples from various datasets and projects and performed translations using popular LLMs including GPT-4 [11], Llama 2 [8], Wizard Vicuna 13B [10], Airoboros 13B [1], StarCoder [24], CodeGeeX [46], and CodeGen [31]. They reported that LLMs were largely ineffective in translating real-world projects, highlighting the need for improvement in code translation techniques.

2.2 Code Translation Benchmarks

Lu et al. [26] unveiled CodeXGLUE, a benchmark dataset designed to assess machine learning models across 10 different program understanding and generation tasks. This dataset, compiled from a variety of open-source projects, includes a specialized code-to-code translation dataset featuring 11,800 Java to C# translation pairs. In a similar vein, Puri et al. [35] introduced CodeNet, a diverse dataset that covers 55 programming languages. CodeNet primarily focuses on code translation tasks, with its data sourced from two prominent online judge platforms: AIZU [2] and AtCoder [3].

Zhu et al. [49] contributed to this growing field with CoST, a parallel code corpus that encompasses seven languages and facilitates snippet-level alignment through code comment matching. CoST is versatile, supporting a range of tasks including code translation, summarization, and synthesis. Building upon CoST, Zhu et al. further introduced XLCoST [48], a dataset offering alignment at both snippet and program levels. Both CoST and XLCoST were curated from GeeksForGeeks [6], a resource-rich website featuring a plethora of data structures and algorithm problems, along with solutions in up to seven popular programming languages.

More recently, Jiao et al. [20] developed G-TransEval, a benchmark that integrates various existing benchmarks into a unique benchmark by categorizing code translation pairs into four types:

[1] https://github.com/RISElabQueens/forge24-code-translation

syntax level, library level, semantic level, and algorithm level translations.

2.3 Prompt Engineering in SE

The remarkable ability of LLMs for zero-shot and few-shot prompting has gained growing interest in exploring how in-context learning and prompt engineering can improve LLM-based applications within SE. Researchers have leveraged LLMs and prompt engineering for various ASE tasks [16, 18, 19, 25, 32].

Gao et al. [18] conducted a study to understand the impact of the choice, order, and number of demonstration examples on the effectiveness of in-context learning in code intelligence tasks. Pan et al. [32] introduced an iterative prompting method that integrates additional contextual information to improve LLM-based code translation. This method enriches the input to language models with comprehensive context, including the original code, prior prompts, erroneous translations, detailed error information, translation instructions, and expected outcomes. Li et al. [25] explored ChatGPT's capability in identifying test cases that reveal bugs in source code. Initially, ChatGPT showed limited success, but with careful prompting, its performance improved significantly. Feng et al. [16] utilized prompts that exploit few-shot learning and chain-of-thought reasoning for LLM-based bug reproduction. Geng et al. [19] delved into the efficacy of LLMs in generating code comments with various intents, focusing on their attributes. They employed the in-context learning approach and designed new strategies for selecting and re-ranking examples to optimize performance.

3 Study Setup

This section discusses the selection of LLMs for our study (Section 3.1), the data collection procedure (Section 3.2), and the utilized prompt templates (Section 3.3).

3.1 Selected Large Language Models

A diverse range of models exists in the rapidly evolving field of LLMs for code generation, offering a variety of capabilities and performance. Recognizing the importance of evaluating these models, we study 11 instances of instruct-tuned decoder-only LLMs, including previously studied and novel models. We select instruct-tuned LLMs to the detriment of their associated base models, as instruct-tuned models are fine-tuned to follow prompted instructions more effectively. Table 1 summarizes the basic information of selected LLMs. The studied models can be grouped across four distinct model families:

- Magicoder [43]: This open-source collection of LLMs, trained on 75K synthetic instruction data using OSS-Instruct, includes the Magicoder-CL and Magicoder-S-CL variants.
- WizardCoder [27]: Building on the StarCoder framework [24], WizardCoder generates intricate code instruction data, with versions ranging from 1B to 34B in size.
- CodeLLama [39]: A family of models based on Llama 2, specialized in code generation, with 7B, 13B, and 34B parameters.
- Mixtral [28]: A Mixture-of-Experts model from Mistral AI, known for its strong performance in code generation.

Our selection of model families for this study is driven by two key factors: their demonstrated strong performance in code generation tasks and their open-source nature, which enables accessibility and collaborative development. Additionally, our study acknowledges the computational demands of running code translation applications based on LLMs. Therefore, we have prioritized models that align well with efficient deployment frameworks, such as vLLM [22], without precluding their compatibility with other platforms like the HuggingFace Text Generation Interface [45]. This consideration ensures that our chosen models are not only high-performing but also practically feasible for widespread use in research and industry applications.

Table 1: Summary of LLMs Used in Current Study

Model Name	Size (Billions)	Context Length (Tokens)	Release Date
CodeLlama Instruct	7	16,384	Aug, 2023
CodeLlama Instruct	13	16,384	Aug, 2023
CodeLlama Instruct	34	16,384	Aug, 2023
Magicoder-S-CL	7	16,384	Dec, 2023
Magicoder-CL	7	16,384	Dec, 2023
WizardCoder	1	8,192	Aug, 2023
WizardCoder	3	8,192	Aug, 2023
WizardCoder Python	7	8,192	Aug, 2023
WizardCoder Python	13	8,192	Aug, 2023
WizardCoder Python	34	8,192	Aug, 2023
Mixtral 8x7B Instruct v0.1	46.7	8,192	Dec, 2023

3.2 Dataset and Preprocessing

We utilize programs from the dataset provided by Pan et al. [32], that are derived from the CodeNet dataset. This dataset contains 1,000 programs in five PLs (C, C++, Go, Java, and Python), 200 programs and test cases for each programming language. There are 20 possible translation combinations of source-target program language pairs considering the five PLs found in the dataset. Therefore, we create 4,000 translation samples that consist of programs in the source programming language and the 4 desired output languages (e.g., one program in C can be translated to C++, Go, Java, and Python).

We use the tokenizer of each of the eleven models to tokenize the input code (program) from each of the code pairs. We can observe in Figure 2 that 97.9% of the programs have a source code token length of up to 3,072 tokens. Therefore, we utilize this number as the cut-off for code pairs, filtering out from our dataset code pairs that have an input program token length greater than 3,072 tokens. In addition, we allow models to generate up to 2,048 new tokens during inference. Doing so ensures that we can fit the input code, prompt, and output into the context window of all models.

After excluding input code exceeding 3,072 tokens in length, our dataset was reduced to 3,912 translation samples. However, the filtering process resulted in an imbalanced dataset due to varying numbers of code pairs for each source-target language combination. As a result, the number of code translation samples across some PL pairs is greater than that of other PL pairs. We then balance the number of code pairs in each source-target language combination by down-sampling those with a higher frequency. In the end, we have a dataset of 3,820 code pairs, with 191 code pairs and test cases for each of the 20 source-target language combinations. Figure 3 showcases the distribution of the final dataset.

3.3 Prompt Templates

We utilize three prompt templates in our case study, as seen in Fig. 4. The first template is proposed by Pan et al. [32] for evaluating open-source LLMs. We refer to it as "Reference Prompt". The "Vanilla

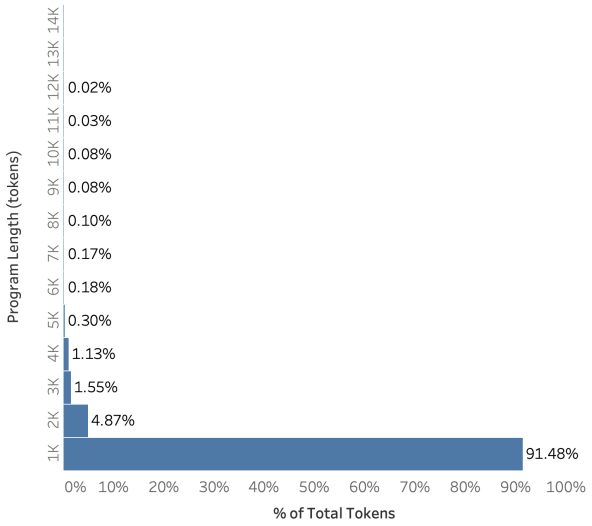

Figure 2: Distribution of program length in Pan et al.'s dataset before the cut-off.

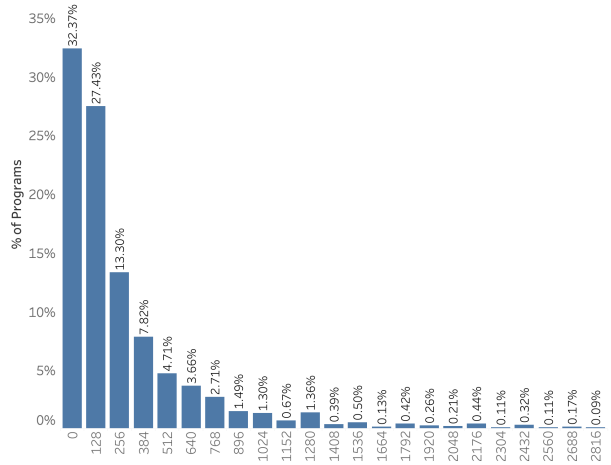

Figure 3: Distribution of the token lengths of programs in our dataset.

Prompt" prompt template, created by us, is tailored for each model based on the model's recommended template from its respective paper or HuggingFace model card.

Additionally, for the instruction in our prompt template, we follow OpenAI's suggested prompting strategy, i.e., "Ask the model to adopt a persona" [2]. This strategy is also adopted in recent instruct-tuned code LLMs, such as Magicoder [43].

Lastly, *Controlled Prompt* is a variation of *Vanilla Prompt* designed with the goal of controlling the output format of the models. The detailed design concerns for this prompt can be found in Section 5.

[2]https://platform.openai.com/docs/guides/prompt-engineering/strategy-give-models-time-to-think

3.4 Evaluation Metrics

In our case study, we adopt execution-based metrics, similar to those used by Pan et al. [32], as we believe they represent the most effective method to assess the model's ability to produce accurate translations. LLMs can generate valid translations that are different from the human-written reference. Text-based metrics can be misleading when evaluating translated code against the reference, as the two pieces of code could achieve a high score but still be functionally different [47]. In our study, we are interested in showcasing the capabilities of the models to generate functionally equivalent programs rather than syntactically similar ones.

Specifically, we considered the following four evaluation metrics when comparing selected LLMs.

Computational Accuracy (CA): Rozière et al. [37] introduced the CA metric, which evaluates whether a transformed target function generates the same outputs as the source function when given the same inputs. The target function is correct if it gives the same output as the source function for every tested input value. In our case, the programs in the dataset are functions that receive input from stdin and produce an output.

Compilation Rate (CR): The number of programs that successfully compiled over the total number of programs in the datasets.

Match Success Rate (MSR): The number of generated outputs where a regular expression designed to extract source code from the output successfully finds a match, over the total number of generated outputs.

Code Extraction Success Rate (CSR): The number of generated outputs for which we are able to extract source code, over the total number of generated outputs.

MSR and CSR are new evaluation metrics that we developed based on our newly proposed method for controlling the format of output, that is explained in Section 5.

3.5 Implementation Details

We begin by downloading the official checkpoints for all evaluated models from Hugging Face. Subsequently, the vLLM framework is employed to conduct inference using Greedy Decoding on four Nvidia RTX 6000 GPUs. Using Greedy Decoding guarantees the reproducibility of the inference outputs across multiple runs. For models ranging from 1 to 7B in size, a single GPU is utilized. Models with 13B employ Tensor Parallelism across two GPUs, while those exceeding 13B leverage the same technique across all four GPUs. Regarding the execution and compilation of programs, our environment is equipped with the following versions of compilers and run-times: OpenJDK 11, Python 3.11, g++ 12, gcc 12, and Go 1.19.

4 RQ1: *What are the characteristics of the output formats across LLMs and prompts?*

Examining the output formats generated by the benchmarked models is a crucial aspect of validating our hypothesis. Additionally, this evaluation sheds light on the prevalent output formats across various models and inspires the development of approaches that can control the output format.

Reference Prompt

```
{input_lang} Code:


{input_code}


Translate the above {input_lang}
code to {output_lang}.


{output_lang} Code:
```

Vanilla Prompt

```
### Instruction:
You are a skilled software
developer proficient in multiple
programming languages. Your task is
to re-write the input source code.
Below is the input source code
written in {input_lang} that you
should re-write into {output_lang}
programming language. You must
respond with the {output_lang}
output code only.

Source code:
{input_code}

### Response:
```

Controlled Prompt

```
### Instruction:
You are a skilled software
developer proficient in multiple
programming languages. Your task is
to re-write the input source code.
Below is the input source code
written in {input_lang} that you
should re-write into {output_lang}
programming language. You must
respond with the {output_lang}
output code only.

Source code:
```
{input_code}
```
### Response:
```
```

**Figure 4: Example of the Prompt Templates used WizzardCoder. The same instruction is used for all the LLMs, following the recommended prompt template from the authors of each LLM. The exception is Reference Prompt, which is used as-is.**

## 4.1 Approach

Our initial step involved creating a representative sample of our dataset to examine the distribution and variety of output formats produced by these models. This was achieved through stratified sampling across source-target programming language combinations within the dataset. As a result, we acquired a subset of 360 code translation pairs, spanning 20 source-target language pairs, which accurately represents our dataset.

For each of the 11 models, we utilized both the Reference and Vanilla Prompt templates (as illustrated in Fig. 4) to generate inference results. With each prompt-model producing 360 inference outcomes, the total number of inferences across all models amounted to 3,960 for each prompt.

To identify and summarize patterns within these outputs, we conducted open-coding as guided by the method described by Stol et al. [40]. To ensure a comprehensive analysis, we further extracted a representative subset of these output formats for each prompt. This involved stratified sampling across the language pairs and LLMs, leading to a subset of 440 entries that reflects the broader set of 3,960 results.

Finally, we estimated the proportions of these identified patterns within the inference results generated from our dataset. This process not only helps in understanding the behavior of different LLMs in response to varying prompts, but also contributes to the broader understanding of LLM output formats in the context of code translations.

## 4.2 Results

We analyze the output formats generated by models in response to two different prompt templates, i.e., the Reference and Vanilla Prompt templates. The models produce outputs in three primary formats in terms of the source code:

(1) **Direct Output**: This format consists solely of source code without any special formatting.
(2) **Wrapped Code**: Here, the source code is enclosed within a code block, delineated by triple back-ticks.
(3) **Unbalanced Back-ticks**: In this format, the source code is followed by a closing triple back-tick, but lacks an opening one.

For each of these output formats, we further categorize the results based on the presence or absence of natural language elements, such as notes, comments, or explanations, accompanying the source code:

- **No additional text**: The inference results contain only source code, with no additional natural language content.
- **Additional text**: Natural language is present in the inference results. This may appear before, after, or interspersed within the code.

Figure 1 illustrates three sample outputs from our dataset, along with their corresponding sub-categories.

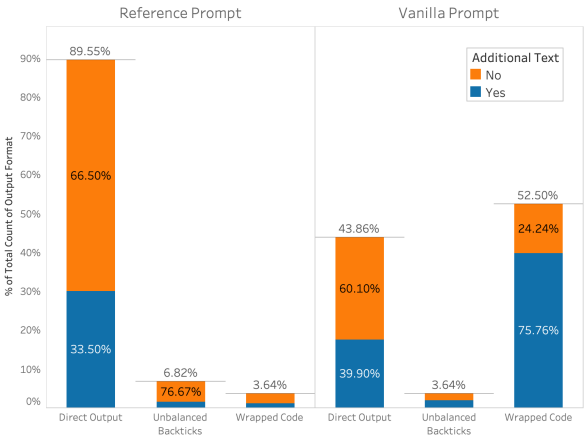


**Figure 5: Distribution of output formats observed for each prompt in RQ1. The height of the bar represents the observed proportion of output formats over the sampled generation outputs for the prompt.**

**Finding 1. The models' output format is not consistent across our dataset, and post-processing is required to extract the**
```

source code in up to 73.6% of the outputs. Among the six combinations (three source code formats, with or without *Additional text*), only when the models generate the *Direct Output* format without *Additional text*, the inference output can be directly employed for evaluation. This is because it solely consists of source code and thus necessitates no further post-processing. As shown in Figure 5, in the case of our *Vanilla Prompt*, only 26.36% (43.86% × 60.10%) of the generated outputs contain solely source code and are immediately usable (over the total 43.86% that are Direct Output, we observe that 60.10% have no additional text, as seen in Fig. 5). The remaining 73.64% require post-processing due to added natural text such as explanations, comments, and notes, or being in a different output format than what we expect (Wrapped Code, Unbalanced Back-Ticks).

Conversely, for our Reference Prompt, we estimate that up to 59.5% (89.55% × 66.50%) of the outputs are suitable for direct evaluation without any post-processing. This percentage represents the outputs in *Direct Code Output* format that do not contain additional text. However, the remaining 40.5% of the outputs will cause compilation (or interpretation) errors due to added comments or being in other output formats than the assumed one.

Finding 2. Different prompts have different output format distributions As shown in Figure 5, the proportions of the three categories of output formats vary based on the design of the prompt. In total, 89.55% of the outputs by the Reference Prompt have a Direct Code format. On the other hand, the Vanilla Prompt outputs mostly *Wrapped Code* format (in 52.50% of the inferences). This result shows that not only do the models generate different amounts of *Additional Text* alongside source code in the Direct Output category (as observed in Finding 1), but also the predominant category of output format varies across prompts.

Finding 3. During inference, models disregard the instruction to *output code only* 59.3% of the time. *Vanilla Prompt* instruct the models to "Output code only" as seen in Fig. 4. We find that in 59.32% of the cases across all the output formats, the model added comments alongside the source code (as derived from Fig. 5). Surprisingly, this number is higher than the *Reference Prompt* that does not instruct the model to output only code. In the later, only 32.73% of the generated output across all output formats have additional text alongside the source code.

RQ1 Summary: We observe that the models generate outputs in varying formats and distributions, depending on the prompt used. In extensive studies comparing different models or prompts, this variation in output formats presents a challenge in understanding the distribution of possible output formats and accurately capturing code. This variation significantly affects the reliability of the metrics derived from the generated output of the models.

5 RQ2: *To what extent can the output format of LLMs be controlled using prompt engineering and lightweight post-processing?*

In RQ1 (Section 4), we confirm that LLMs can generate outputs with different formats. Thus, we aim to explore the extent to which the output format of LLMs can be controlled (i.e., shifting the output distribution to one preferred type) so that we can extract code automatically in a unified way.

5.1 Approach

Our approach to control the models' output format uses a combination of prompt engineering and regular expression (regex) parsing.

Our prompt engineering method attempts to improve the consistency of the output format across all the models, such that the generated code in the output can be directly extracted for the highest number of programs as possible, without capturing comments that would interfere with the compilation or interpretation of the code. For prompt engineering, we modify the *Vanilla Prompt* (Section 3.3) by adding a control statement that instructs the model to generate the code excerpt within triple back-ticks (Wrapped Code format). Figure 4 (c) shows the derived *Controlled Prompt*.

We use the Regex in Fig. 6 to find and extract source code from the generated outputs. The Regex is designed to match the *Wrapped Code* format (i.e. The text inside the first block delimited by three back-ticks on the output). The regex also ignores, if present, the first line in the output that often denotes the programming language the code is written into as it can be seen in Figure 1.

```
Response:?.*?```(?:(?:java|cpp|csharp|python|c|go|
C\+\+|Java|Python|C#|C|Go))?(.+)```
```

Figure 6: Regex designed to match and extract source code from code blocks (Wrapped Code format) in the generated output text.

We obtain a subset of 440 samples through stratified sampling across the language pairs and LLMs, following the same methodology described in RQ1 (Section 4.1) and estimate the proportion of each of the output formats. We estimate our Regex's accuracy (Match Success Rate) on this subset of 440 samples.

For the cases where there was a match, we analyze what percentage of those was actually source code or not (Precision and Recall). We also manually analyze the cases where there was no match, and explain the reasons for such mismatches.

5.2 Results

Figure 7 shows the distribution of output format types between the *Vanilla Prompt* and the *Controlled Prompt*.

Finding 4. The output format can be controlled through a combination of prompt engineering and lightweight post-processing, achieving a Match Success Rate of 95.45% and a Code Extraction Success Rate (CSR) of 92.73%. Our proposed prompt is capable of steering the output across models, and our proposed Regex (Fig. 6) matches 95.45% of the generated outputs. As the matched content could be source code with added text, we manually examine the matches to confirm what proportion of the matches are source code only. We observe that out of the 95.45% there is 2.73% of matches that are not source code. In other words, we achieve a Code Extraction Success Rate (CSR) of 92.73%.

Next, we perform a detailed analysis on the 2.73% of matches that do not consist of source code. Through this analysis, we discover that the content matched comprises natural language text rather than source code. More interestingly, 100% of those cases are caused

by the Mixtral 8x7B model. Specifically, we observe the emergence of a new output format we name **Re-Wraps Code** generated by the Mixtral 8x7B model, as seen in Figure 1 (d). In the *Re-Wraps Code* format, the model does not complete the code block opened using back-ticks in the *Controlled Prompt*, ignoring our output control mechanism. Instead, the model opens a new set of back-ticks, adds source code and finishes by closing the back-tick group. Because our Regex is designed to capture *Wrapped Code*, when it is applied to the Re-Wrapped Code format it matches from the beginning of the back-ticks we introduced in the *Controlled Prompt* up to the first set of back-ticks generated by the model, not capturing the source code.

RQ2 Summary: The output format can be controlled to increase the percentage of output that can be matched by the Regex from 52.50% in the Vanilla Prompt to 95.45% in the Controlled Prompt. Controlling the output significantly helps capture more source code.

6 RQ3: *What's the impact of output control on the reported performance of LLMs?*

In RQ2, we show the feasibility of controlling output formats of LLMs for code translation via a combination of prompt engineering and lightweight post-processing regex. As the MSR changes, we hypothesize that commonly considered evaluation metrics for code translation (i.e., CR and CA) should also change. Thus, in this RQ we empirically quantify what is the impact of our output control on the reported performances of LLMs on the whole 3,820 translation pairs.

6.1 Approach

To answer this question, we leverage the *Vanilla Prompt* and *Controlled Prompt* that are shown in Figure 4. Using each of the prompts, we perform translation on all the 3,820 code pairs in our dataset. We employ various combinations of prompts and source code extraction methods to explore their effects on the results reported by three execution-based metrics. The combinations evaluated are:

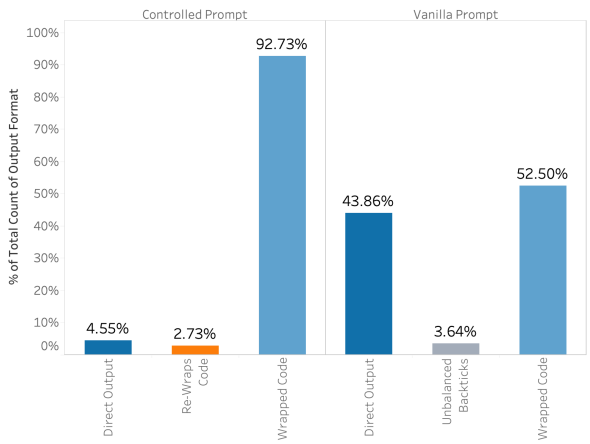

Figure 7: Comparison of the distribution of output formats of the Controlled Prompt (left side) vs Vanilla Prompt (right side).

(1) **Vanilla + Direct Evaluation (VDE):** We utilize the Vanilla Prompt and directly evaluate the generated output. We do not perform any regex matching or source code extraction.
(2) **Vanilla + Regex (VRE):** We utilize the Vanilla Prompt and apply our regex to extract source code.
(3) **Controlled + Regex (CRE):** We utilize the Controlled Prompt and apply our regex to extract source code.

6.2 Results

Table 2 shows the Match Success Rate (MSR), Compilation Rate (CR), and Computational Accuracy (CA) for the eleven considered models using three considered scenarios (VDE, VRE, and CRE) on the 3,820 code translation pairs.

Finding 5: The integration of prompt engineering with a lightweight post-processing technique (CRE) demonstrated superior MSR, CR, and CA among the tested combinations. The CRE approach achieved an impressive average MSR of 93.40%, coupled with an average CR of 52.19% and an average CA of 35.74% over the models.

The MSR and CR values for VDE and VRE, as detailed in Table 2, corroborate the insights gained from our manual analysis in RQ2. These findings highlight the efficacy of our output control method in enhancing the likelihood of extracting potentially compilable source code from inference output, thereby boosting the CR. In contrast, neglecting the output format and attempting direct compilation of inference outputs (VDE) achieves the lowest average CR at just 7.95%. This stark difference underscores the critical role of the output format and control when reporting and interpreting the performance of LLMs for code translation.

The three combinations (CRE, VDE, VRE) share similar patterns in terms of CA with the other two metrics (MSR, CR). The lowest CA value is 4.92% for VDE. The second highest CA is for VRE, with 35.74%. Even though the Vanilla Prompt is not designed to increase the chances of outputting a specific format, it shows higher CR and CA than VDE.

Additionally, it can be observed that Magicoder models and WizzardCoder 34B had a CR of 0% in VDE, with no generation output that can be compiled. To further examine the possible causes of this phenomenon, we randomly sampled 350 generated outputs for each of WizzardCoder 34B, Magicoder-CL, and Magicoder-S-CL models. Our inspection confirms that all the 350 samples for each of the models contain *Additional text* or an output format different than *Direct Code*, which explains why none of the generated outputs are compilable.

Lastly, we can observe that the CR of Mixtral 8x7B decreased from 53.51% in VRE to 0% on CRE, which is the opposite of what we would expect in CRE. Manual inspection confirms that this is because 100% of the 350 generated out samples are in another format than the one expected by our regex, such as *Re-Wraps Code* or others. This showcases the importance of carefully inspecting the output format of each model when comparing them in a benchmark.

Finding 6: The consideration of output formats can significantly alter the outcomes when benchmarking various LLMs. In the context of CRE, our analysis reveals a noteworthy trend: WizzardCoder Python 34B emerges as the top-performing model, boasting an average CA of 43.85% across all languages. This finding

Table 2: Results for different combinations of prompt and extraction method across eleven LLMs. VDE refers to (Vanilla + Direct Evaluation), VRE refers to (Vanilla + Regex), and CRE refers to (Controlled + Regex) combinations.

		Match Success Rate (MSR)			Compilation Rate (CR)			Average CA		
Model Name	**Model Size**	**CRE**	**VRE**	**VDE**	**CRE**	**VRE**	**VDE**	**CRE**	**VRE**	**VDE**
CodeLlama Instruct	7	94.2%	6.60%	–	50.42%	2.77%	**30.03%**	33.27%	1.83%	18.30%
CodeLlama Instruct	13	96.9%	0.68%	–	57.70%	0.05%	27.20%	38.12%	0%	**18.51%**
CodeLlama Instruct	34	98.1%	40.16%	–	49.29%	22.80%	7.93%	34.06%	13.17%	5.18%
Magicoder-CL	7	88.5%	67.98%	–	56.47%	47.20%	0%	37.33%	32.02%	0%
Magicoder-S-CL	7	99.0%	**99.35%**	–	**68.09%**	**68.80%**	0%	43.30%	**43.12%**	0%
Mixtral 8x7B	46.7	60.4%	82.20%	–	0%	53.51%	2.17%	0%	35.60%	1.28%
WizardCoder	1	96.8%	72.91%	–	47.39%	48.72%	0.13%	18.61%	17.25%	0.03%
WizardCoder	3	96.9%	68.09%	–	59.45%	53.74%	3.90%	32.28%	29.92%	0.55%
WizardCoder Python	7	98.3%	40.65%	–	55.29%	29.35%	15.79%	34.11%	15.79%	10.13%
WizardCoder Python	13	98.9%	71.28%	–	62.93%	47.80%	0.31%	36.20%	27.64%	0.1%
WizardCoder Python	34	**99.4%**	28.48%	–	66.49%	18.38%	0%	**43.85%**	15.42%	0%
Average	–	**93.40%**	**52.58%**	–	**52.19%**	**35.74%**	**7.95%**	**31.92%**	**20.98%**	**4.92%**

is particularly striking when juxtaposed with the performance of Magicoder-S-CL, a relatively smaller model with 7B parameters, which impressively secures the second-highest CA at 43.30% across all models. This suggests that model size might not be the sole determinant of effectiveness in the code translation task, same as reported in prior work [43]. Conversely, in the VDE scenario, where output control is ignored, a different pattern is observed. Here, both the CodeLlama Instruct 7B and 13B models demonstrate superior CA compared to others. This variance in the measured performance underlines the significant impact that the output format and control can have when reporting the capabilities of different LLMs.

RQ3 Summary: The characteristics of the output format and its control significantly influence the Computational Accuracy (CA) metric. The lowest average CA, at 4.92%, is achieved when the generated output from the LLMs is compiled directly (VDE). The highest average CA across all models, at 31.92%, is obtained using the Controlled Prompt with Regex (CRE) . We conclude that careful inspection and source code extraction based on the output format of the models is necessary to achieve comparable performance results.

7 Discussion

Our initial key finding underscores the challenge posed by the inconsistency in output formats across various large language models (LLMs), regardless of the input prompt provided. This necessitates a focused effort on standardizing or regulating output formats to enable more meaningful comparisons between models. Consequently, it is essential to bring this issue to the attention of researchers and practitioners. Overlooking this issue can result in the effective utilization of only a fraction of the benchmark dataset, ultimately impacting the validity of model evaluations. Notably, researchers and practitioners should not assume that LLMs will consistently output in a Direct Output format or another desired output format across models. Therefore, researchers must acknowledge output variability to avoid under-utilizing benchmark datasets and should explicitly state their assumptions about the models' output format when reporting experimental design and evaluation results.

Our results also show that a combination of prompt engineering and lightweight post-processing regex can effectively control the output formats and achieve a remarkable MSR of 93.40% as seen in Table. 2. This simple approach improves the average Compilation Rate from 35.74% in VRE to 93.40% in CRE. Our controlled prompt is mainly effective in reducing noise from natural language that would otherwise cause a compilation or interpretation error, emphasizing the potential for prompt engineering to enhance successful code extraction rates. In addition, our study advocates for low-cost techniques such as prompt engineering and regular expressions to address inconsistent output issues. These techniques go beyond benchmarking and can be applied as a post-processing step in code translation applications. Therefore, in addition to researchers engaging in code translation studies, practitioners can benefit from implementing controlled prompts, as this technique improves the percentage of captured code from the LLMs' outputs, providing better end-user results. This insight emphasizes the importance of prompt engineering as a cost-effective strategy to address the inconsistency in output formats.

We additionally examined how controlling the output format affects CA. Our findings demonstrate a significant enhancement in CA, with the CRE combination resulting in an average CA of 31.92% across all models, as opposed to the lowest average of 4.92% when directly compiling outputs from LLMs (VDE). Therefore, controlling the output format emerges as a crucial factor in the evaluation of code translations. We emphasize the need to consider output format and extraction method as a significant parameter in evaluating LLMs for code translation.

We also highlight the potential existence of internal mechanisms in certain models, such as Mixtral, that could influence the steerability of output formats. Researchers are advised to be aware of such models and adapt code extractors accordingly, underlining the importance of model-specific considerations in the evaluation process.

Our case study, especially RQ2 and RQ3, is grounded in a specific prompt template design, i.e., the Vanilla Prompt. Similar to other LLM-based ASE models [43], we can not claim the effectiveness of

our output control for all potential prompt templates. However, our primary aim is not to deliver a universal output control solution, i.e., a specific prompt template paired with a fixed regular expression — applicable to every prompt for every model, nor is it to introduce an enhanced LLM-based code translation method. Our objectives are twofold: First, we aim to underscore the considerable impact that inconsistent output formats have on benchmarking LLMs for code translation, thereby increasing awareness of this issue. Second, we intend to present one solution that effectively tackles this problem. Furthermore, we believe that our analytical and design methodology can be adapted to other prompts and models, which could allow for a more accurate measurement of the true performance of LLMs for code translation.

8 Threats to Validity

External Validity. We acknowledge several limitations that might impact the external validity of our findings. They are mainly introduced by the selection of the dataset, target LLMs, and considered prompts. Firstly, our research relies on a subset of the CodeNet benchmark. While this may limit the diversity of the programs (source code) and target PLs, CodeNet is a comprehensive and widely recognized benchmark in related work, and our selection of PLs is based on their popularity. Such a setting is also considered in a recent related work [32]. This subset offers a diverse range of programming problems and solutions (we considered five popular programming languages and corresponding 20 PL translation cases), which we believe enhances the relevance and applicability of our findings. Regarding our model selection, we limited our study to 11 instruct-tuned LLMs from four model families. Although this represents a constrained subset of available LLMs, these were carefully chosen for their popularity and recent advancements in the field. We choose instruct-tuned models rather than base models as instruct-tuned models are specifically optimized to follow instructions more effectively. This means they are more likely to produce outputs that are closely aligned with the given prompts.

Results in RQ3 show that our current specific prompt and regex may not apply to all models, which is particularly evident in the case of the Mixtral 8X7B model. While this highlights a limitation in the universality of the proposed specific output control methods across different models, our design method, i.e., how we analyze the output format and control it using a combination of prompt and regex, can be generalized and easily adapted to other models.

Internal Validity. The main threats to internal validity are introduced by the design of the prompt template and the labeling of output format types. The prompt templates used in our experiments were custom-designed, which may not represent the optimal choice for each model. However, we carefully crafted these prompt templates based on templates recommended on the models' official sites. Regarding the labeling of output formats, this task was undertaken by the first author of this paper. While this could introduce subjective bias or errors in labeling and taxonomy, we established clear, easily identifiable categories to minimize ambiguity and enhance consistency in our categorization process.

Construct Validity. We consider dynamic execution-based metrics to benchmark LLMs for code translation. Dynamic execution-based evaluation is used in recent studies on bench-marking LLMs for code translation [20, 32] and code generation tasks [15]. Unlike text-oriented metrics, such as the BLEU score, which measures the overlap between the tokens in the translation and the reference, execution-based metrics capture the semantic equivalence of the code translation. Moreover, we believe that simply ignoring the inconsistent output format issue would also lead to a less precise calculation of BLEU if one directly assumes the output is the target code and compares it with the reference. This could be validated by future research.

In this paper, as the CodeNet benchmark contains one test case per sample, there may be translations considered computationally accurate, but that do not contain identical functionality to the input program, i.e., false positives. This may influence the results reported in Table 2 on RQ3. However, our main goal is to point out the importance of considering output format control when utilizing LLMs for code translation, and such a conclusion can be supported by other metrics, such as extraction success rate and compilation rate, which are not sensitive to the quality of the test cases.

9 Conclusion

In this study, we conducted an empirical analysis of the output formats generated by 11 popular instruct-tuned large language models (LLMs) in code translation tasks, using 3,820 translation pairs. Our findings reveal significant variability in the output formats of these LLMs. Despite employing different prompts that adhere to recommended strategies from official LLM sites, we observed that the generated output from the models included additional text or presented the code in quoted or partially quoted forms, adversely affecting the compilation rate. To address this challenge, we proposed an output control approach, which combines a specific prompt design with regex to extract code from the output. Our case study on the 3,820 translation pairs shows that this method enhanced the Match Success Rate from 52.58% to 93.40% across the models. Notably, this approach increased the average Computational Accuracy from 4.92% to 31.92% across the models.

Our research highlights the importance of the output format, prompt engineering, and code extraction methods in the evaluation of LLMs for code translation. The varying behaviors of different models in generating output formats, coupled with their sensitivity to prompt structures, can significantly influence key evaluation metrics such as Computational Accuracy and Compilation Rate. Overlooking these aspects could lead to misinterpretations and unfair comparisons among LLMs. We believe our insights enrich benchmarking practices in code translation, offering practitioners another perspective on how to measure the vast potential that LLMs hold for code translation.

10 Acknowledgments

We express our gratitude to the Natural Sciences and Engineering Research Council of Canada (NSERC) for their support, with funding reference number RGPIN-2019-05071. Additionally, we extend our appreciation to the Vector Institute for its offering of the Vector Scholarship in Artificial Intelligence, which was awarded to the first author. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of Huawei and/or its subsidiaries and affiliates.

References

[1] 2024. Airoboros 13B HF fp16. hhttps://huggingface.co/TheBloke/airoboros-13B-HF Accessed: date-of-access.

[2] 2024. Aizu online judge. https://onlinejudge.u-aizu.ac.jp/ Accessed: date-of-access.

[3] 2024. Atcoder. https://atcoder.jp/ Accessed: date-of-access.

[4] 2024. C to Go translator. https://github.com/gotranspile/cxgo Accessed: date-of-access.

[5] 2024. C2Rust. https://github.com/immunant/c2rust Accessed: date-of-access.

[6] 2024. GeeksForGeeks. https://www.geeksforgeeks.org/ Accessed: date-of-access.

[7] 2024. Java 2 CSharp Translator for Eclipse. https://sourceforge.net/projects/j2cstranslator/ Accessed: date-of-access.

[8] 2024. LLama 2. https://ai.meta.com/research/publications/llama-2-open-foundation-and-fine-tuned-chat-models/ Accessed: date-of-access.

[9] 2024. Sharpen - Automated Java->C coversion. https://github.com/mono/sharpen Accessed: date-of-access.

[10] 2024. Wizard-Vicuna-13B-Uncensored float16 HF. https://huggingface.co/TheBloke/Wizard-Vicuna-13B-Uncensored-HF Accessed: date-of-access.

[11] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).

[12] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. 2015. *Using machine translation for converting python 2 to python 3 code.* Technical Report. PeerJ PrePrints.

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

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

[15] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533* (2023).

[16] Sidong Feng and Chunyang Chen. 2023. Prompting Is All Your Need: Automated Android Bug Replay with Large Language Models. *arXiv preprint arXiv:2306.01987* (2023).

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

[18] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, and Michael R Lyu. 2023. Constructing Effective In-Context Demonstration for Code Intelligence Tasks: An Empirical Study. *arXiv preprint arXiv:2304.07575* (2023).

[19] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. (2024).

[20] Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. [n. d.]. On the Evaluation of Neural Code Translation: Taxonomy and Benchmark. ([n. d.]).

[21] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 173–184.

[22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. https://doi.org/10.48550/arXiv.2309.06180 arXiv:2309.06180 [cs].

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

[24] 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. 2023. StarCoder: may the source be with you! https://doi.org/10.48550/arXiv.2305.06161 arXiv:2305.06161 [cs].

[25] Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 14–26.

[26] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[27] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. https://doi.org/10.48550/arXiv.2306.08568 arXiv:2306.08568 [cs].

[28] Mistral AI Team. 2023. *Mixtral of Experts.* https://mistral.ai/news/mixtral-of-experts/ Accessed: 2024-01-13.

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

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

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

[32] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2023. Understanding the Effectiveness of Large Language Models in Code Translation. http://arxiv.org/abs/2308.03109 arXiv:2308.03109 [cs] (Accepted by ICSE 2024).

[33] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2001. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02*. Association for Computational Linguistics, Philadelphia, Pennsylvania, 311. https://doi.org/10.3115/1073083.1073135

[34] Maja Popović. 2015. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, Ondřej Bojar, Rajan Chatterjee, Christian Federmann, Barry Haddow, Chris Hokamp, Matthias Huck, Varvara Logacheva, and Pavel Pecina (Eds.). Association for Computational Linguistics, Lisbon, Portugal, 392–395. https://doi.org/10.18653/v1/W15-3049

[35] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. https://doi.org/10.48550/arXiv.2105.12655 arXiv:2105.12655 [cs].

[36] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. http://arxiv.org/abs/2009.10297 arXiv:2009.10297 [cs].

[37] 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.

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

[39] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, 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. 2023. Code Llama: Open Foundation Models for Code. https://doi.org/10.48550/arXiv.2308.12950 arXiv:2308.12950 [cs].

[40] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 120–131. https://doi.org/10.1145/2884781.2884833

[41] Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2023. Code Translation with Compiler Representations. https://doi.org/10.48550/arXiv.2207.03578 arXiv:2207.03578 [cs].

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

[43] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source Code Is All You Need. https://doi.org/10.48550/arXiv.2312.02120 arXiv:2312.02120 [cs].

[44] Justin D. Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I. Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. 2021. Perfection Not Required? Human-AI Partnerships in Code Translation. In *26th International Conference on Intelligent User Interfaces*. ACM, College Station TX USA, 402–412. https://doi.org/10.1145/3397481.3450656

[45] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. HuggingFace's Transformers: State-of-the-art Natural Language Processing. https://doi.org/10.48550/arXiv.1910.03771 arXiv:1910.03771 [cs].

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

[47] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. http://arxiv.org/abs/2302.05527 arXiv:2302.05527 [cs].

[48] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474* (2022).

[49] Ming Zhu, Karthik Suresh, and Chandan K Reddy. 2022. Multilingual code snippets training for program translation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 11783–11790.