Enhancing Code Translation in Language Models with Few-Shot Learning via Retrieval-Augmented Generation

Manish Bhattarai
Theoretical Division
Los Alamos National Laboratory
Los Alamos, NM
ceodspspectrum@lanl.gov

Javier E. Santos
Earth & Environmental Science Division
Los Alamos National Laboratory
Los Alamos, NM
jesantos@lanl.gov

Shawn Jones
Computer, Computational & Statistical Sciences
Los Alamos National Laboratory
Los Alamos, NM
smjones@lanl.gov

Ayan Biswas
Computer, Computational & Statistical Sciences
Los Alamos National Laboratory
Los Alamos, NM
ayan@lanl.gov

Boian Alexandrov
Theoretical Division
Los Alamos National Laboratory
Los Alamos, NM
boian@lanl.gov

Daniel O'Malley
Earth & Environmental Science Division
Los Alamos National Laboratory
Los Alamos, NM
omalled@lanl.gov

***Abstract*—The advent of large language models (LLMs) has revolutionized the field of code translation, enabling automated translation between programming languages. Despite these advancements, the accuracy and reliability of these models often falter in complex translation tasks due to a lack of contextual understanding. This paper introduces a novel approach to enhance code translation through Few-Shot Learning augmented with retrieval-based techniques. By leveraging a repository of existing code translations, we dynamically retrieve the most relevant examples to guide the model in translating new code segments. Our method, based on Retrieval-Augmented Generation (RAG), significantly improves translation quality by providing contextual examples that the model can learn from in real-time. We chose RAG over traditional fine-tuning methods due to its ability to leverage existing codebases or a locally stored corpus of code, allowing it to dynamically adapt to diverse translation tasks without the need for extensive retraining. Extensive experiments on diverse datasets, using open LLM models such as Starcoder, Llama3-70B Instruct, CodeLlama-34B Instruct, Granite-34B Code Instruct, and Mixtral-8x22B, and commercial LLM models such as GPT-3.5 turbo, and GPT-4o demonstrate the superiority of our approach over traditional zero-shot, particularly in translating between Fortran and C++.We also explored different numbers of shots (examples provided to the model during inference) — specifically 1, 2, and 3 shots — and various embedding models for RAG, including Nomic-Embed, Starencoder, and CodeBERT, to evaluate the robustness and effectiveness of our approach.**

***Index Terms*—code translation, large language models, retrieval augmented generation, few shot learning, fortran, C++**

I. Introduction

The rapid evolution of programming languages and the need to maintain legacy codebases have created a substantial demand for automated code translation tools. Traditional approaches to code translation involve extensive manual effort and expertise in both the source and target languages. With the rise of large language models (LLMs), such as GPT-3 [1], Codex [2], and CodeBERT [3], automated code translation has become increasingly feasible. These models leverage vast amounts of training data to generate code snippets in various languages, demonstrating impressive capabilities in general language understanding and generation tasks.

However, despite these advancements, the performance of language models in code translation tasks remains inconsistent, particularly when dealing with complex or less common programming constructs. A significant challenge lies in the model's ability to comprehend and generate code that adheres to the syntactic and semantic rules of both the source and target languages. Zero-shot and few-shot learning techniques (where ”shots” refer to the number of examples provided to the model during inference) have shown promise [4], but they often lack the depth of contextual understanding required for high-fidelity translations.

Fine-tuning language models for specific code translation tasks is a common approach to improve performance [5]. However, fine-tuning requires substantial computational resources and time, and it may not generalize well to new or unseen tasks. In contrast, Retrieval-Augmented Generation (RAG) offers a dynamic alternative by leveraging a repository of existing code translations to provide contextual examples during the translation process [6]. This method allows the model to adapt to various tasks without extensive retraining, making it a more flexible and efficient solution.

In this paper, we propose a RAG framework that enhances Few-Shot Learning for code translation tasks. Our approach involves maintaining a repository of code translation examples and dynamically retrieving the most relevant examples based on the input code segment. By providing the model with multiple contextual examples, RAG facilitates a deeper understanding of the translation task, leading to more accurate and reliable code generation. For this, an embedding model, which converts code snippets into numerical vectors that capture their semantic meaning, is used.

We conducted extensive experiments using various open language models, including Starcoder [7], Llama3-70B Instruct [8], CodeLlama-34B Instruct [9], Granite-34B Code Instruct [10], Mistral 8x22B [11] and Codestral [12] and commercial models such as GPT-3.5 [1] and GPT-4o on publicly available Fortran-C++ translation datasets. We evaluated the performance of our approach with different numbers of shots (1, 2, 3) and various embedding models, such as Nomic-Embed [13], Starencoder [7], and CodeBERT [3]. For datasets that did not have direct translation pairs, we performed pairwise comparisons to assess the translation consistency across different LLMs.

The remainder of this paper is structured as follows: Section 2 reviews related work in code translation and few-shot learning. Section 3 outlines our RAG methodology, including the retrieval mechanism and integration with language models. Section 4 presents our experimental setup and results, followed by a discussion in Section 5. Finally, Section 6 concludes the paper and highlights future research directions.

II. Related Works

The field of code translation, particularly from Fortran to C++, has seen various innovative approaches and methodologies aimed at improving translation accuracy and efficiency. One notable contribution is the development of a high-performance computing (HPC) code translation dataset [5]. This dataset pairs OpenMP Fortran with C++ code snippets, facilitating the training and evaluation of machine learning models for effective code translation. The dataset's quality was ensured through human-level evaluations by expert programmers, which significantly refined the translations by assessing correctness, readability, and semantic retention of the original Fortran code.

In addition to dataset creation, efforts have been made to re-engineer legacy Fortran code into maintainable C++ code. A study explored the maintainability of translated Fortran code by evaluating various software quality metrics [14]. The research concluded that re-engineered Fortran to C++ code exhibited high maintainability standards, making it a viable solution for modernizing legacy systems while ensuring the translated code remains efficient and easy to maintain.

Automated tools have also played a crucial role in facilitating Fortran to C++ translation. The CFortranTranslator, an open-source tool, converts Fortran90/Fortran77 code to C++14 while maintaining the abstraction level of the original code [15]. This tool effectively handles mixed fixed-form and free-form Fortran code, providing a practical solution for developers needing to translate Fortran code to leverage modern C++ features and frameworks; however, the resulting code may not always be easy to read or maintain, posing long-term challenges.

The advent of LLMs has further revolutionized code translation. A study titled "Lost in Translation" emphasized the importance of providing contextual information, such as stack traces and error messages, to improve LLM-based code translation accuracy [16]. This iterative prompting approach demonstrated significant improvements in translation success rates by incorporating additional informative context into the prompts. The study also provided a comprehensive taxonomy of bugs introduced by LLMs during code translation, offering valuable insights into common error modalities and mitigation strategies.

Fine-tuning LLMs for domain-specific tasks has been another area of active research [17]. Prompt-oriented fine-tuning using Low-Rank Adaptation (LoRA) [18] has been proposed to adapt LLMs to specific programming languages and domains. This method leverages domain-specific vocabulary and demonstrates improved translation quality by efficiently fine-tuning pre-trained models for specific tasks [5]. In the paper, the authors demonstrated that finetuning the LLM improved the translation performance by 9 folds.

III. Methods

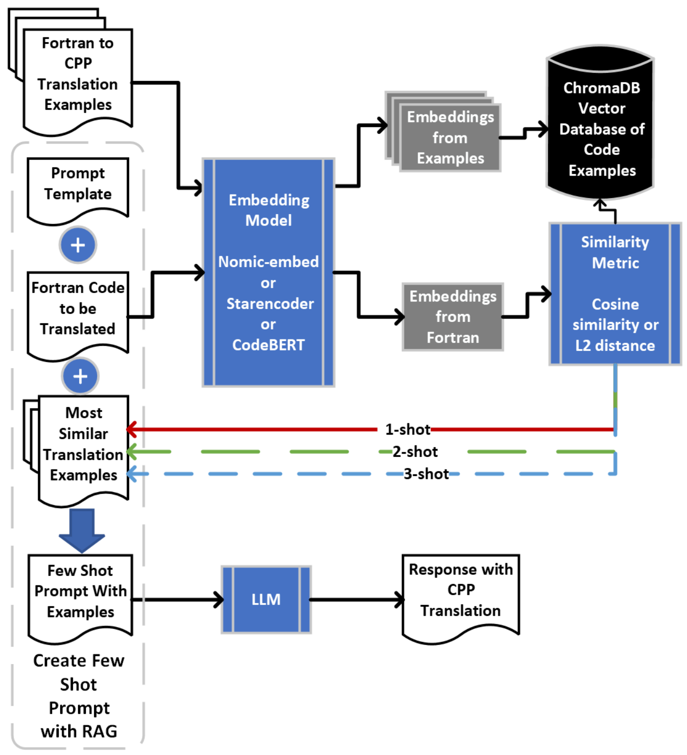

Fortran to CPP Translation Examples
Prompt Template
Fortran Code to be Translated
Most Similar Translation Examples
Embedding Model
Nomic-embed or Starencoder or CodeBERT
Embeddings from Examples
ChromaDB Vector Database of Code Examples
Embeddings from Fortran
Similarity Metric
Cosine similarity or L2 distance
1-shot
2-shot
3-shot
Few Shot Prompt With Examples
LLM
Response with CPP Translation
Create Few Shot Prompt with RAG

Fig. 1: Pipeline for creating a few shot prompt through RAG for code translation.

Our study presents a RAG based pipeline designed to enhance the accuracy and contextual understanding of automated code translations from Fortran to C++, as illustrated in Figure 1. This method integrates LLMs with retrieval mechanisms, enabling the generation of high-quality translations through dynamically provided contextual examples. The pipeline is highly adaptive, allowing users to plug in different LLMs, embedding models, datasets, and adjust the number of shots for evaluating translation performance. Models can be seamlessly loaded directly from Hugging Face [19], specific directories tailored to each model type, or via API calls as specified by user input. For retrieval with RAG, either l_2 distance or $cosine$ similarity metric can be used.

Dataset Preparation

We leveraged three datasets to evaluate our models:

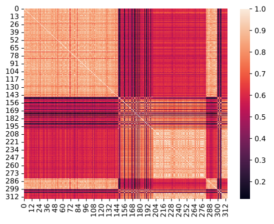
1) Numerical Recipes Dataset:: This dataset comprises pairs of Fortran and C++ code snippets, ensuring a robust set of examples [20]. Each code pair is meticulously curated to maintain high standards of quality and relevance. To ensure quality, we standardized code style, removed comments, handled whitespace and special characters, and mapped Fortran subroutines to their C++ equivalents. One downside of this dataset is that it relies on a specific library of functions, which may limit its general applicability. This dataset comprises 298 Fortran-C++ pairs.

2) HPC Fortran2CPP Dataset:: This dataset was derived from [5].The dataset comprises comprehensive Fortran to C++ translation pairs and was meticulously curated from the NAS Parallel Benchmarks (NPB), Polyhedral Benchmark (PolyBench), and DataRaceBench (DRB) repositories. The NPB dataset evaluates supercomputer performance with computational fluid dynamics benchmarks, PolyBench provides programs for polyhedral compilation research, and DRB includes OpenMP programs for data race detection tool evaluation. The authors performed standardized code style as we did for Numerical Recipe dataset and additional calibration was done using similarity tests ensured semantic fidelity, while expert programmers conducted human-level evaluations to assess correctness, readability, and semantic retention. This dataset comprises 315 Fortran-C++ pairs.

3) Stack-V2 Dataset:: The Stack V2 dataset is a comprehensive collection of code samples sourced from a wide range of repositories on GitHub, focusing on high-performance computing and various computational problems [21]. This dataset includes approximately half a million Fortran code snippets, providing a diverse set of examples for robust evaluation. For our study, we sampled 500 Fortran examples from this dataset by selecting files with lengths between 1000 and 10,000 bytes from unique repositories, prioritizing those with the highest combined star and fork event counts to ensure high-quality and diverse samples. Since Stack-V2 doesn't have Fortran-C++ pairs, we extracted the files containing metadata, codes, and comments. We then leveraged Llama3-70B Instruct model to extract the executable Fortran code, discarding other metadata. This cleaned dataset was subsequently used to compare translation performance across various LLMs in a zero-shot setting.

Embedding Generation and Example Retrieval

A crucial component of our RAG pipeline is the generation of embeddings for each Fortran code snippet. We employ various embedding models such as Nomic-Embed [13], Starencoder [7], and CodeBERT [3] to produce these embeddings, which are essential for assessing translation performance. These embeddings capture the semantic essence of the code snippets, enabling efficient retrieval of the most contextually relevant examples from the dataset. For efficient vector storage and retrieval, we leverage ChromaDB, which stores embeddings along with the corresponding source and target codes. We evaluated the retrieval performances with l_2 and $cosine$ similarity metrics.


(a)

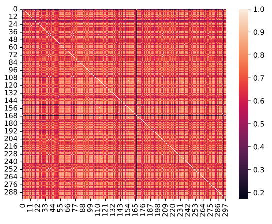

(b)

Fig. 2: Similarity of dataset embeddings for a)HPC Fortran2CPP dataset and b) Numerical Receipe dataset based on Nomic Embed model

Once the embeddings are generated, the system dynamically retrieves the most similar examples based on cosine similarity or Euclidean distance. This process ensures that the model receives the most relevant examples tailored to each specific translation task, thereby enhancing the accuracy and efficiency of the code translation. The retreived translation pairs are then augmented with the original query and directed to LLM for translation. Figure 2 displays the cosine similarity between the Fortran source code embeddings corresponding to the Numerical Recipes and HPC Fortran2CPP datasets. The similarity matrices provide valuable insights into the structural and functional relationships within and between the datasets.

In the HPC Fortran2CPP dataset (Figure 2a), we observe distinct clusters of high similarity, indicating that certain groups of Fortran code snippets share significant semantic and syntactic features. These clusters likely correspond to common computational patterns or routines that are frequently reused within the dataset. The clear block-diagonal structure suggests that the HPC Fortran2CPP dataset contains several distinct modules or subcomponents that exhibit high internal coherence. In contrast, the Numerical Recipes dataset (Figure 2b) presents a more uniformly distributed similarity matrix with fewer pronounced clusters. This distribution implies a more diverse set of code snippets with varying functionalities and less repetition of specific patterns. The relatively consistent similarity across the matrix suggests that while individual snippets may not be as closely related, there is a broad underlying similarity in the type of computational problems addressed by the Numerical Recipes dataset.

Comparing the two datasets, we can infer that the HPC Fortran2CPP dataset might be more modular and specialized,

```
System: You are adept at translating Fortran code
    into CPP with high accuracy, ensuring that all
    syntax, semantics, and specific language
    features are correctly and efficiently converted
User: Translate the following code from Fortran to
    CPP:
{Fortran code snippet}
```

Fig. 3: Zero-Shot Translation Prompt Template

with specific sections of code dedicated to particular tasks. In contrast, the Numerical Recipes dataset appears to be more general-purpose, covering a wide range of computational routines without strong modular separation. These insights are crucial for understanding the nature of the datasets and for accessing the RAG performance for code translation, as they highlight the varying degrees of internal consistency and the potential challenges in translating diverse code structures.

Few-Shot Learning with Retrieval-Augmented Generation

In-context learning for source code translation leverages the power of few-shot examples to improve the performance of language models in generating accurate translations. In a typical zero-shot setting, a language model M generates a target translation T for a given query Q (source code) directly, which can be mathematically represented as $T = M(Q)$. However, the performance of M can be significantly enhanced by conditioning it on a set of k example pairs of source and target code, $\{(S_i, T_i)\}_{i=1}^{k}$, before generating the translation for Q. This few-shot learning process can be expressed as:

$$T = M(\{(S_i, T_i)\}_{i=1}^{k}, Q).$$

In a RAG setup, this process is further optimized by incorporating a retrieval mechanism R that selects the most relevant k example pairs from a large corpus $\mathcal{C}$ based on the query Q. The retrieval step can be mathematically formulated as:

$$\{(S_i, T_i)\}_{i=1}^{k} = R(Q, \mathcal{C}).$$

Subsequently, the model M generates the translation T using the retrieved example pairs and the query, expressed as:

$$T = M(R(Q, \mathcal{C}), Q).$$

The translation process begins with a zero-shot approach, where the model translates the Fortran code to C++ without any additional context. We then generate embeddings for the Fortran code snippets using models like Nomic-Embed, Starencoder, and CodeBERT. These embeddings capture the semantic essence of the code, facilitating efficient retrieval of the top-k most relevant examples from ChromaDB, our vector storage and retrieval system.

For Zero-shot comparisons, we apply the zero-shot translation prompt template (Figure 3) with the Fortran code to be translated. For few-shot prompts, we use the prompt template

```
System: You are adept at translating Fortran code
    into CPP with high accuracy, ensuring that all
    syntax, semantics, and specific language
    features are correctly and efficiently converted
User: Here's an example of a code translated from
    Fortran to CPP:
Here's the Fortran code:
{Similar Fortran code example}
Here's the CPP translation:
{Corresponding CPP translation}
[Repeat for k examples]
Now translate the following code from Fortran to CPP
    :
{Fortran code snippet}
```

Fig. 4: Few Shot Translation Prompt Template

shown in Figure 4 with the rest of the pipeline. The model processes the input Fortran code along with the retrieved examples. The tokenized input is fed into the model, which generates the corresponding C++ code. This approach ensures that the model receives the most relevant examples tailored to each specific translation task, thereby enhancing translation accuracy and efficiency.

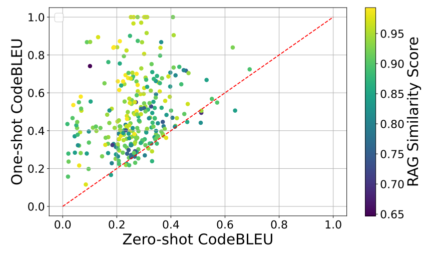
Evaluation and Experimental Setup

The generated translations are evaluated using the CodeBLEU metric [22], which assesses the quality of code translations by considering both syntactic and semantic correctness. CodeBLEU extends the traditional BLEU metric by incorporating four key components: *N-gram Match Score*, which measures the precision of n-grams in the translated code compared to the reference code, ensuring the retention of original token sequences; *Weighted N-gram Match Score*, which enhances the n-gram match score by weighting different n-grams based on their importance, thus focusing on critical code patterns and structures; *Syntax Match Score*, which evaluates the syntactic correctness of the translated code, ensuring adherence to the grammatical rules of the programming language; and *Dataflow Match Score*, which assesses the semantic correctness by analyzing the data flow within the program, ensuring the preservation of logical flow and functional equivalence.

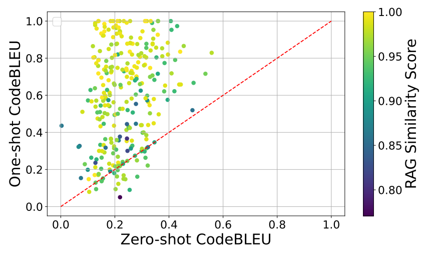
Our experiments involved extensive testing with various combinations of models, shot numbers, RAG retrieval metrics, and embedding models. We utilized Nomic-Embed, Starencoder, and CodeBERT as embedding models. For code translation, we leveraged open LLMs such as Starcoder, Llama3-70B Instruct, Code LLaMA-34B, Granite-34B Code Instruct, and Mistral 8x22B, as well as commercial LLM models such as GPT-3.5 turbo and GPT-4o/GPT-4 turbo. We evaluated the performance of the models for zero, one, two, and three shots. Each experiment aimed to measure the improvements in translation quality facilitated by the RAG approach using CodeBLEU as the primary evaluation metric.

IV. Results and Discussions

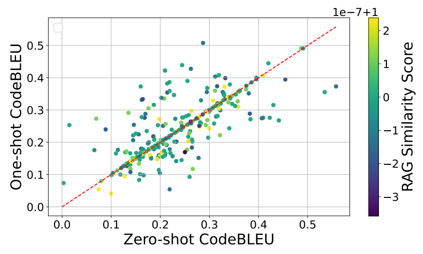
1) Performance Across Models and Embeddings: The results highlight significant variability in performance across



(a)

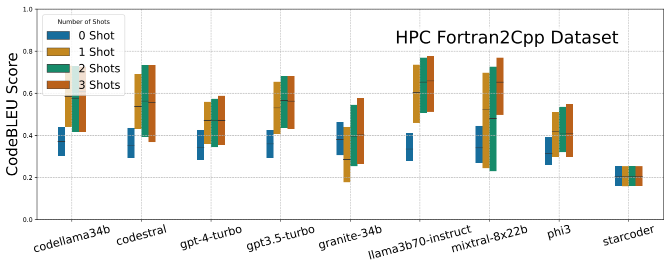


(b)

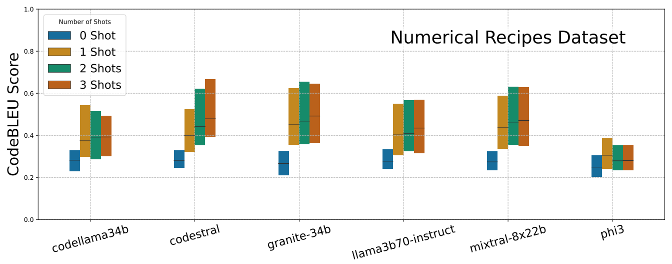


(c)

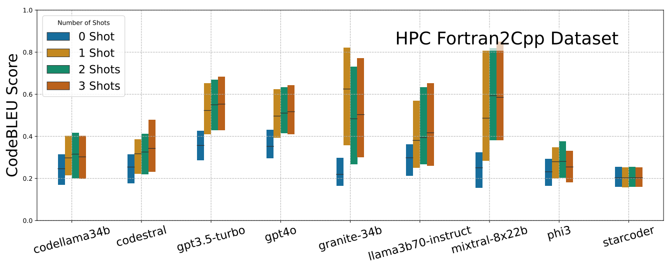
Fig. 5: Performance Comparison of One-shot vs. Zero-shot in the RAG Pipeline Using the Nomic-embed Embedding Model across Various Models and Datasets: (a) Granite-34B Code Instruct on the Numerical Recipes Dataset, (b) Granite-34B Code Instruct on the HPC Fortran2CPP Dataset, and (c) Granite-34B Code Instruct on the HPC Fortran2CPP Dataset with bad RAG setup (utilize largest distance metric as retreival) . The color of each data point represents the similarity of the retrieved one-shot example pair to the query Fortran code, with the legend indicating the intensity range of the similarity metric. Generally, a higher similarity score correlates with a higher CodeBLEU score.



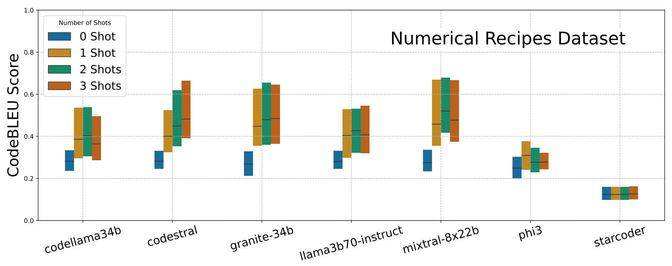
(a) HPC Fortran2CPP dataset with cosine similarity



(b) Numerical Recipes dataset with cosine similarity



(c) HPC Fortran2CPP dataset with l_2 distance



(d) Numerical Recipes dataset with l_2 distance

Fig. 6: Overview of CodeBLEU metrics for code translation with different LLMs using cosine similarity and l_2 distance for RAG retrieval on (a) and (c) HPC Fortran2CPP dataset and (b) and (d) Numerical Recipes dataset.

different models and embedding strategies. The breakdown of Codebleu metric for different models in Zero-shot setting is shown in Table I. The table provides key insights into the performance of different models on zero-shot tasks using the Nomic-embed embedding model with l_2 distance. Notably, GPT-4 Turbo and GPT-3.5 Turbo achieved the highest zero-shot CodeBLEU scores of 0.371 and 0.367, respectively, indicating their strong initial performance without additional context. These models also scored highly on Syntax Tree and Dataflow metrics, which measure syntactic correctness and dataflow consistency, respectively, highlighting their ability to generate structurally sound and semantically accurate code. In contrast, Granite-34B and Llama3-70B Instruct, while having slightly lower CodeBLEU scores (0.237 and 0.309, respectively), showed balanced performance across all metrics, signifying their robust handling of various translation aspects. Metrics like Ngram and Weighted Ngram, which assess exact token matches and weighted token matches, respectively, were particularly low for StarCoder, reflecting its struggles with precise code generation in zero-shot tasks. Mixtral-8x22B exhibited high variance in the Numerical Recipes dataset, especially in CodeBLEU and Ngram scores, suggesting inconsistency in its translation quality.

However in the Few-shot task, the Granite-34B Code Instruct, Llama3-70B Instruct and Mixtral-8x22B model consistently outperformed others across all embedding types and

TABLE I: Mean with standard deviation scores for CodeBLEU metrics of Zero Shot prompts for different LLMs, ordered by CodeBLEU performance

Dataset	**Model**	**CodeBLEU**	**Ngram**	**Weighted Ngram**	**Syntax Tree**	**Dataflow**
HPC Fortran2CPP	GPT-4o	0.371 ± 0.002	0.188 ± 0.001	0.290 ± 0.001	0.495 ± 0.003	0.504 ± 0.005
	GPT-3.5 Turbo	0.367 ± 0.001	0.199 ± 0.003	0.301 ± 0.004	0.483 ± 0.002	0.474 ± 0.004
	Llama3-70B Instruct	0.309 ± 0.001	0.135 ± 0.002	0.225 ± 0.002	0.325 ± 0.005	0.512 ± 0.003
	Codestral-22B	0.245 ± 0.000	0.090 ± 0.001	0.155 ± 0.001	0.158 ± 0.002	0.520 ± 0.002
	CodeLlama-34B Instruct	0.243 ± 0.000	0.090 ± 0.001	0.166 ± 0.001	0.169 ± 0.002	0.490 ± 0.001
	Mixtral-8x22B (176B)	0.241 ± 0.001	0.059 ± 0.001	0.104 ± 0.001	0.140 ± 0.001	0.331 ± 0.001
	Granite-34B Code Instruct	0.237 ± 0.001	0.084 ± 0.001	0.145 ± 0.001	0.171 ± 0.000	0.500 ± 0.002
	Phi-3 3.8B	0.228 ± 0.000	0.063 ± 0.004	0.119 ± 0.006	0.165 ± 0.001	0.501 ± 0.002
	StarCoder 15.5B	0.206 ± 0.000	0.009 ± 0.000	0.014 ± 0.000	0.127 ± 0.000	0.636 ± 0.001
Numerical Recipes	Codestral-22B	0.288 ± 0.000	0.042 ± 0.000	0.101 ± 0.000	0.512 ± 0.000	0.497 ± 0.000
	Llama3-70B Instruct	0.283 ± 0.001	0.036 ± 0.000	0.103 ± 0.001	0.508 ± 0.003	0.484 ± 0.002
	CodeLlama-34B Instruct	0.281 ± 0.002	0.035 ± 0.001	0.108 ± 0.003	0.482 ± 0.007	0.493 ± 0.003
	Granite-34B Code Instruct	0.272 ± 0.002	0.063 ± 0.001	0.109 ± 0.001	0.462 ± 0.003	0.443 ± 0.003
	Mixtral-8x22B (176B)	0.264 ± 0.021	0.077 ± 0.015	0.140 ± 0.013	0.340 ± 0.070	0.385 ± 0.052
	Phi-3 3.8B	0.249 ± 0.002	0.036 ± 0.001	0.085 ± 0.002	0.432 ± 0.002	0.438 ± 0.004
	StarCoder 15.5B	0.133 ± 0.001	0.007 ± 0.000	0.010 ± 0.000	0.269 ± 0.000	0.244 ± 0.002

TABLE II: Delta in Mean CodeBLEU scores between Zero- and Few-Shot prompts using the Nomic-embed Embedding Model with l_2 Distance for HPC Fortran2CPP dataset. Our RAG method for choosing examples improves CodeBLEU scores for most of the LLMs listed.

			Δ in CodeBLEU scores		
Dataset	**Model**	**Zero-shot**	**1-shot**	**2-shot**	**3-shot**
HPC Fortran2CPP	CodeLlama-34B Instruct	0.243	+0.078	+0.084	+0.069
	Codestral-22B	0.245	+0.074	+0.105	+0.158
	GPT-3.5 Turbo	0.367	+0.157	+0.176	+0.188
	GPT-4o	0.371	+0.132	+0.153	+0.153
	Granite-34B Code Instruct	0.237	**+0.363**	+0.278	+0.302
	Llama3-70B Instruct	0.309	+0.117	+0.137	+0.151
	Mixtral-8x22B (176B)	0.241	+0.288	**+0.355**	**+0.367**
	Phi-3 3.8B	0.228	+0.058	+0.070	+0.050
	StarCoder 15.5B	0.206	0.000	0.000	0.000
Numerical Recipes	CodeLlama-34B Instruct	0.281	+0.145	+0.157	+0.147
	Codestral-22B	0.288	+0.148	+0.207	+0.250
	Granite-34B	0.272	**+0.229**	+0.240	+0.254
	Llama3-70B Instruct	0.283	+0.149	+0.163	+0.168
	Mixtral-8x22B (176B)	0.264	+0.227	**+0.264**	**+0.291**
	Phi-3 3.8B	0.249	+0.068	+0.051	+0.044
	StarCoder 15.5B	0.133	0.000	0.000	0.000

learning configurations, achieving the highest CodeBLEU scores. For instance, under the nomic-embed model embedding, Granite-34B Code Instruct achieved a zero-shot CodeBLEU of 0.237 and improved to 0.6 in the one-shot setting for HPC Fortran2CPP dataset with l_2 norm as shown in Table II. This demonstrates the model's strong capability in understanding and translating Fortran code into C++ with minimal context.

While we conducted experiments using various embedding models, our findings indicate that Nomic-embed and Starencoder exhibited equivalent performance in few-shot settings. However, CodeBERT consistently underperformed compared to the other two models. For instance, CodeLlama-34B Instruct achieved a zero-shot CodeBLEU of 0.243 with Nomic-embed, which improved to 0.321 in the two-shot configuration. In contrast, CodeBERT's performance did not show comparable improvement. This discrepancy can be attributed to the maximum token limit of each embedding model; CodeBERT has a token limit of 512, whereas both Starencoder and Nomic-embed support up to 8192 tokens. Given the longer length of Fortran codes to be translated, the limited token capacity of CodeBERT likely hindered its performance. To ensure clarity and focus, we present results using Nomic-embed throughout the text, as they are consistent with those obtained using Starencoder.

2) Impact of Few-Shot Learning: Few-shot learning consistently enhanced translation performance across all models and embeddings. The progressive improvement from one-shot to three-shot learning configurations was evident, with models like Granite-34B Code Instruct, Llama3-70B Instruct, and Mixtral-8x22B showing substantial gains as shown in Table II. For instance, Granite-34B Code Instruct with nomic_embed improved from a zero-shot CodeBLEU of 0.24 ± 0.09 to 0.60 ± 0.27 in the one-shot setting, further improving to $0.540.27$ and 0.54 ± 0.21 in the two-shot and three-shot settings, respectively. Thus, the RAG setup played a significant role in enhancing code translation quality. By leveraging a retrieval mechanism to provide relevant context from similar code snippets or documentation, RAG setups help models to better understand the structure and semantics of the source code. This additional context is especially beneficial in few-shot learning scenarios, where providing relevant examples can significantly improve the translation quality.

Figure 5 illustrates the performance comparison of one-shot versus zero-shot scenarios in the RAG pipeline using the Nomic-embed embedding model across various models and datasets. Specifically, the comparisons are made for Figure 5a Granite-34B Code Instruct on the Numerical Recipes Dataset, Figure 5b Granite-34B Code Instruct on the HPC Fortran2CPP Dataset, and Figure 5c Granite-34B Code Instruct on the HPC Fortran2CPP Dataset with a poorly configured RAG setup (utilizing the largest distance metric as retrieval). The color of each data point represents the similarity of the retrieved one-shot example pair to the query Fortran code, with the legend indicating the intensity range of the similarity metric. Generally, the plots demonstrate that a higher similarity score between the retrieved example and the query correlates with a higher CodeBLEU score. When the utilized retrieval metric is altered, this brings a significant change in performances. To further validate this, we have leveraged l_2 distance and cosine similarity as retrieval metric. Figure 6c and 6d showcases model performances leveraging l_2 metric whereas Figure 6a and 6b showcases performance with cosine similarity. This relationship underscores the effectiveness of leveraging similar examples in enhancing the performance of the RAG pipeline, particularly in one-shot scenarios.

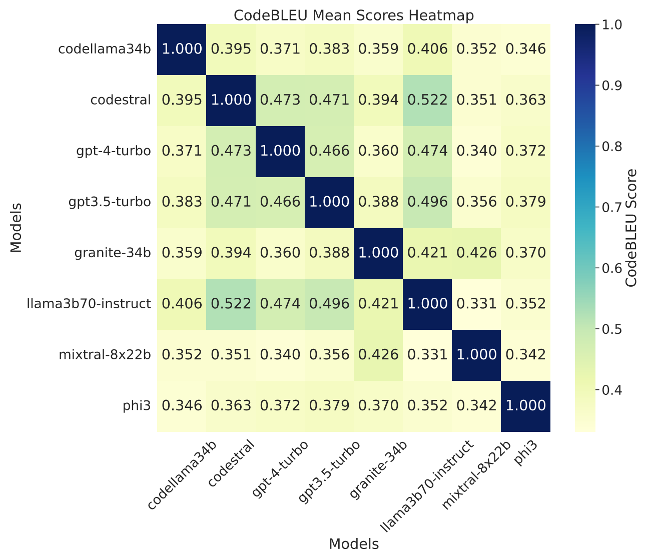


Fig. 7: Pairwise zeroshot performance comparison between models for translated Stack-V2 dataset

3) Dataset-Specific Performance: HPC Fortran2CPP vs. Numerical Recipes in Few-Shot setting: The datasets used in this study, namely HPC Fortran2CPP and Numerical Recipes, exhibited different performance characteristics across the models. The HPC Fortran2CPP dataset generally yielded higher CodeBLEU scores compared to the Numerical Recipes dataset. For instance, Granite-34B Code Instruct with CodeBERT embedding achieved a one-shot CodeBLEU of 0.6 on the HPC Fortran2CPP dataset, whereas it scored 0.49 ± 0.20 on the Numerical Recipes dataset. This discrepancy can be attributed to the inherent differences in the complexity and structure of the code in these datasets. The HPC Fortran2CPP dataset, which may contain more standardized and less complex code, allowed models to perform better. In contrast, the Numerical Recipes dataset, with potentially more intricate and varied code structures, posed greater challenges for the models.

4) Impact of LLM models: When comparing different models, Mixtral-8x22B, Llama3-70B, and Granite-34B stood out as the top performers in few-shot settings, whereas GPT-3.5 Turbo and GPT-4 Turbo excelled in zero-shot settings. Starcoder, on the other hand, showed relatively lower performance, particularly in the Zero-shot setting, with a CodeBLEU of 0.21. Additionally, Starcoder did not show significant improvement in one-shot settings, likely due to its smaller context length.

Models such as llama3, Codestral, Mixtral, Granite, and CodeLlama consistently outperformed others across multiple metrics, particularly in terms of CodeBLEU scores. This superior performance can be attributed to their explicit pre-training or fine-tuning for code-related tasks. These models have likely been trained on extensive code-specific data and tasks, enabling them to better understand and generate programming languages. Conversely, models like Phi-3 did not perform as well in comparison, likely because they were not specifically optimized for code-related tasks. While Phi-3 may have been trained on a diverse set of texts, including some programming language data, their training was not as focused on code-specific tasks as Granite and CodeLlama. Consequently, their ability to handle Fortran to C++ translation is less robust.

While GPT models such as GPT-4 Turbo excel in zero-shot settings with high initial CodeBLEU scores, their performance does not increase substantially with additional shots. This plateau in performance can be attributed to GPT models prioritizing the generation of executable and semantically correct code over strict alignment with ground truth translations. As a result, their incremental benefit from additional examples is limited compared to other models.

Starcoder's notably poor performance can be explained by a couple of factors. First, Starcoder is a smaller model with a smaller token limit, which restricts its capacity to understand and generate complex code structures compared to larger models. Additionally, it appears that Starcoder's training may not have been as focused on code-specific data, leading to lower performance even with multiple shots. This suggests that for tasks requiring a deep understanding and generation of code, model size and the specificity of training data play crucial roles in determining performance.

5) Results on Unlabelled Dataset in zero-shot settings: We first translated the StackV2 Fortran dataset using different LLMs and computed the pairwise CodeBLEU similarity between the model translations. The results are presented in Figure 7. In this experiment, we aimed to evaluate the consistency and similarity of translations produced by various models. The heatmap in Figure 7 illustrates the pairwise CodeBLEU scores, where each cell represents the similarity between translations generated by a pair of models. From the heatmap, we can observe the following key points: The highest pairwise similarity scores are observed between *Codestral* and *Llama3-70B Instruct* models, with a score of 0.522. The *Granite-34B Code Instruct* model shows relatively high similarity with several models, including *Llama3-70B Instruct* and *Mixtral-8x22B*. Models *CodeLlama-34B Instruct*, *GPT 4 Turbo*, and *GPT 3.5 Turbo* also demonstrate a moderate degree of consistency in their translations. These results indicate that while there is a degree of variability in the translations generated by different models, certain models produce more consistent and similar translations compared to others. The main takeaway from this experiment is that identifying models with reliable and consistent translations guides researchers in model selection for future tasks, thereby enhancing translation accuracy, reliability, and advancing automated code translation methodologies.

V. Conclusion and Future Work

In conclusion, the results of this study underscore the importance of model architecture, training data specificity, and the effective use of few-shot learning in enhancing code translation tasks. Models explicitly pre-trained or fine-tuned for code-related tasks, such as GPT-3.5/4 and Llama3-70B-Instruct, demonstrate superior performance, while models with more general training, like Phi-3 3.8B fall short. The findings also highlight the potential limitations of smaller models like Starcoder in handling complex code translation tasks, emphasizing the need for adequate model size and training data to achieve optimal performance. The use of RAG setups further enhances the translation quality by providing relevant context, proving to be a valuable strategy in few-shot learning scenarios. The current limitation in Fortran-C++ pairs challenges fine-tuning LLMs and establishing benchmarks. Our Stack-v2 dataset experiments should guide expert translation collections paired with RAG setups, enhancing model performance and applicability across code translation tasks. Future research will expand the dataset and refine the RAG framework to improve translation quality and reliability.

VI. Acknowledgement

This research was funded by the LANL ASC grant AI4Coding and the LANL Institutional Computing Program, supported by the U.S. DOE NNSA under Contract No. 89233218CNA000001.

References

[1] K. I. Roumeliotis and N. D. Tselikas, “ChatGPT and Open-AI models: A preliminary review,” *Future Internet*, vol. 15, no. 6, p. 192, 2023.

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, “Evaluating Large Language Models Trained on Code,” *arXiv preprint arXiv:2107.03374*, 2021.

[3] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, *et al.*, “CodeBERT: A Pre-Trained Model for Programming and Natural Languages,” *arXiv preprint arXiv:2002.08155*, 2020.

[4] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, “Large Language Models are Zero-Shot Reasoners,” *Advances in neural information processing systems*, vol. 35, pp. 22199–22213, 2022.

[5] B. Lei, C. Ding, L. Chen, P.-H. Lin, and C. Liao, “Creating a Dataset for High-Performance Computing Code Translation using LLMs: A Bridge Between OpenMP Fortran and C++,” in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, IEEE, 2023.

[6] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, *et al.*, “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[7] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, *et al.*, “StarCoder: may the source be with you!,” *Transactions on Machine Learning Research*, 2023.

[8] AI@Meta, “Llama 3 Model Card,” 2024.

[9] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, *et al.*, “Code Llama: Open Foundation Models for Code,” *arXiv preprint arXiv:2308.12950*, 2023.

[10] M. Mishra, M. Stallone, G. Zhang, Y. Shen, A. Prasad, A. M. Soria, M. Merler, P. Selvam, S. Surendran, S. Singh, *et al.*, “Granite Code Models: A Family of Open Foundation Models for Code Intelligence,” *arXiv preprint arXiv:2405.04324*, 2024.

[11] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. de las Casas, E. B. Hanna, F. Bressand, G. Lengyel, G. Bour, G. Lample, L. R. Lavaud, L. Saulnier, M.-A. Lachaux, P. Stock, S. Subramanian, S. Yang, S. Antoniak, T. L. Scao, T. Gervet, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, “Mixtral of Experts,” *arXiv preprint arXiv:2401.04088*, 2024.

[12] M. AI, “Codestral Model Card,” 2024.

[13] Z. Nussbaum, J. X. Morris, B. Duderstadt, and A. Mulyar, “Nomic Embed: Training a Reproducible Long Context Text Embedder,” *arXiv preprint arXiv:2402.01613*, 2024.

[14] Z. G. T. Jr., “Quantifying software maintainability on re-engineered translation of FORTRAN to C++ code,” 2004.

[15] C. Neo, “CFortranTranslator,” 2016. https://github.com/CalvinNeo/CFortranTranslator.

[16] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, “Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2023.

[17] J. Zheng, H. Hong, X. Wang, J. Su, Y. Liang, and S. Wu, “Fine-tuning Large Language Models for Domain-specific Machine Translation,” *arXiv preprint arXiv:2402.15061*, 2023.

[18] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, “Lora: Low-rank adaptation of large language models,” in *International Conference on Learning Representations*, 2021.

[19] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, “Huggingface’s transformers: State-of-the-art natural language processing,” *arXiv preprint arXiv:1910.03771*, 2020.

[20] W. H. Press, W. T. Vetterling, S. A. Teukolsky, and B. P. Flannery, *Numerical Recipes*. Cambridge University Press, 1988.

[21] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, “StarCoder 2 and The Stack v2: The Next Generation,” *arXiv preprint arXiv:2402.19173*, 2024.

[22] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, “CodeBLEU: a Method for Automatic Evaluation of Code Synthesis,” *arXiv preprint arXiv:2009.10297*, 2020.