Enhancing Cross-Language Code Translation via Task-Specific Embedding Alignment in Retrieval-Augmented Generation

Manish Bhattarai[1], Minh Vu[1], Javier E. Santos [2], Ismael Boureima[1], Daniel O' Malley [2],

[1]Theoretical Division, Los Alamos National Laboratory, Los Alamos, NM 87544,
[2] Earth & Environmental Science Division, Los Alamos National Laboratory, Los Alamos, NM 87544

Correspondence: ceodspspectrum@lanl.gov

Abstract

We introduce a novel method to enhance cross-language code translation from Fortran to C++ by integrating task-specific embedding alignment into a Retrieval-Augmented Generation (RAG) framework. Unlike conventional retrieval approaches that utilize generic embeddings agnostic to the downstream task, our strategy aligns the retrieval model directly with the objective of maximizing translation quality, as quantified by the CodeBLEU metric. This alignment ensures that the embeddings are semantically and syntactically meaningful for the specific code translation task. Our methodology involves constructing a dataset of 25,000 Fortran code snippets sourced from Stack-V2 dataset and generating their corresponding C++ translations using the LLaMA 3.1-8B language model. We compute pairwise CodeBLEU scores between the generated translations and ground truth examples to capture fine-grained similarities. These scores serve as supervision signals in a contrastive learning framework, where we optimize the embedding model to retrieve Fortran-C++ pairs that are most beneficial for improving the language model's translation performance. By integrating these CodeBLEU-optimized embeddings into the RAG framework, our approach significantly enhances both retrieval accuracy and code generation quality over methods employing generic embeddings. On the HPC Fortran2C++ dataset, our method elevates the average CodeBLEU score from 0.64 to 0.73, achieving a 14% relative improvement. On the Numerical Recipes dataset, we observe an increase from 0.52 to 0.60, marking a 15% relative improvement. Importantly, these gains are realized without any fine-tuning of the language model, underscoring the efficiency and practicality of our approach.

1 Introduction

Cross-language code translation is a critical task in modern software development, especially as legacy programming languages, such as Fortran, continue to be prevalent in scientific computing, while more contemporary languages like C++ are favored for their performance and versatility in production environments. The goal of automatic translation from Fortran to C++ is to preserve the functionality and structure of legacy code while benefiting from the optimizations and ecosystem of C++. However, achieving high-quality translations that adhere to the syntax and semantic norms of the target language remains a challenging problem, particularly when there is a lack of large, aligned datasets or evaluation metrics that cover both source and target languages effectively.

Traditional approaches to cross-language translation, such as Retrieval-Augmented Generation (RAG) (Lewis et al., 2020) typically involve two phases: first, retrieving relevant examples from a database, followed by a language model generating code conditioned on both the query and the retrieved examples. In prior efforts, the retrieval models in RAG systems have relied on general-purpose embedding models (Bhattarai et al., 2024; Li et al.), which are not tailored to the specific nuances of code translation. These embeddings aim to retrieve relevant pairs from the source and target languages but do not directly optimize for the quality of the generated code. As a result, while the retrieved examples may be relevant in a broad sense, they often fail to guide the language model towards producing translations that maximize fidelity to the ground truth in the target language.

Given the scarcity of high-quality parallel Fortran–C++ data, we generate synthetic C++ translations from abundant Fortran code using an LLM to create a pseudo-parallel corpus. Although these synthetic translations may not be flawless, they provide a robust similarity signal that enables effective alignment of Fortran code embeddings. This "pseudo-alignment" enhances the retrieval of relevant examples in our RAG framework, lead-

Proceedings of the 4th International Workshop on Knowledge-Augmented Methods for Natural Language Processing (KnowledgeNLP'25), pages 107–117

ing to significant improvements in downstream translation quality as demonstrated by a consistent 14–15% gain in CodeBLEU scores. We collect a dataset of 25,000 Fortran code examples from Stack V2 (Lozhkov et al., 2024) and use the LLaMA 3.1-8B (Touvron et al., 2023) model to generate corresponding C++ translations. In the absence of ground truth C++ translations, we evaluate the quality of these translations using pairwise CodeBLEU similarity scores. This metric captures both syntactic correctness and semantic fidelity, providing a robust signal for aligning the retrieval model through contrastive learning.

The proposed approach aims to addresses the shortcomings of general-purpose embedding models by integrating task-specific metrics into the retrieval optimization process. By aligning the retrieval model with the downstream task of producing high-quality C++ code, our method ensures that the examples retrieved during inference are not just broadly similar but are semantically and syntactically aligned in a way that enhances the LLM's generative performance. The result is a significant improvement in translation quality, as measured by CodeBLEU, over previous methods that lack such alignment.

Our contribution is twofold: first, we demonstrate the effectiveness of contrastive learning for fine-tuning retrieval models in the context of cross-language code translation, using a task-specific metric to guide alignment. Second, we show that optimizing retrieval for downstream generation tasks can lead to state-of-the-art results, particularly in cases where aligned datasets are not readily available for both source and target languages. This work not only advances the field of code translation but also opens up new possibilities for applying similar techniques to other language pairs and domains where task-specific evaluation metrics are available for only one side of the translation.

2 Related Work

Historically, code translation strategies before the advent of LLMs relied heavily on rule-based and statistical machine translation (SMT) systems (Koehn, 2009). These systems used predefined rules or statistical mappings between the source and target programming languages, such as tree-based translation approaches that mapped syntax trees between languages. While these methods provided structured and interpretable outputs, they were limited in their ability to handle the semantic complexities of different programming languages and struggled with code diversity, edge cases, and idiomatic translations.

With the rise of deep learning and LLMs, fine-tuning models on large datasets became the go-to method for improving code translation. Models like CodeBERT (Feng et al., 2020) and Codex (Chen et al., 2021), when fine-tuned on specific language pairs, improved translation quality by leveraging vast amounts of parallel code data. However, the main limitation of LLM fine-tuning lies in the resource-intensive process. Fine-tuning requires substantial amounts of labeled data and computational resources, making it impractical for niche or legacy languages like Fortran, where parallel data may be scarce.

As a next step, task-specific alignment of LLMs emerged to improve translation by better guiding the model's output. While alignment techniques help improve output fidelity, they still necessitate fine-tuning or explicit modification of the LLM itself, which can be resource-intensive and may still fall short of generalization when translating between languages with significant structural differences (Mishra et al., 2024).

RAG introduced a more flexible approach by allowing LLMs to retrieve and condition their outputs on example pairs from a relevant dataset. While RAG improves translation by augmenting the model's input, the effectiveness of this strategy depends on the quality and relevance of the retrieved examples. In an example case (Bhattarai et al., 2024), the retrieval step relies on general-purpose embeddings like Nomic-Embed or CodeBERT, which, although effective at retrieving semantically similar code, are not optimized for specific downstream metrics like CodeBLEU. As a result, the LLM might not always retrieve the examples that would best assist in producing translations aligned with target-specific quality metrics.

The approach we propose offers a significant advantage by focusing on semantic alignment of the retrieval mechanism without the need to fine-tune the LLM itself. Through contrastive learning, we optimize the embedding model to retrieve Fortran-C++ pairs that are more likely to maximize the downstream metric (e.g., CodeBLEU) when used by the LLM for generation. This strategy ensures that the most relevant examples are retrieved for each translation task, improving the generation quality without requiring computationally

expensive fine-tuning of the LLM. This retrieval alignment makes RAG more efficient and better suited for translating between languages where high-quality paired datasets may not be available. By concentrating on improving the quality of retrieved examples, our method achieves high-quality translation with minimal additional model training, leveraging existing LLM capabilities more effectively.

3 Methods

This section provides the technical description of our proposed method.

3.1 Problem setting

We consider the standard code translation scenario leveraging a language model G, in which a target translated code c^t of a query source code c^s is generated using G:

$$c^t = G\left(c^s\right) \tag{1}$$

In practice, conditioning G on k example pairs of source and target code $D := \left\{\left(c_i^s, c_i^t\right)\right\}_{i=1}^k$, can significantly enhance translation. This few-shot learning approach can be expressed as: $c^t = G\left(c^s, D\right)$

In a RAG framework, this process is further refined by integrating a retrieval mechanism R that identifies the most pertinent k example pairs from a large corpus $\mathcal{C}$ based on the query c^s. By expressing this retrieval step as $D = R(c^s, \mathcal{C})$, we can describe the conventional translation scenario leveraging G as

$$c^t = G\left(c^s, R(c^s, \mathcal{C})\right) \tag{2}$$

In practice, the input source code for the retrieval are embedded using a neural network Ψ, which are generally agnostic to the downstream task. We denote c_Ψ^s as the embedding of the source code c^s under the embedding Ψ. Hence, Eq. 2 can be expressed as

$$c^t = G\left(c^s, R(c_\Psi^s, \mathcal{C}_\Psi)\right) \tag{3}$$

under the usage of the embedding model Ψ. Here, the notation $\mathcal{C}_\Psi$ refers to the fact that the embedding is applied onto the corpus of c^s.

Some common embedding modules for the retrieval code translation are Nomic-Embed (Nussbaum et al., 2024), StarEncoder (Li et al., 2023), and CodeBERT (Feng et al., 2020). However, as the performance of the translation task heavily depends on the relevance and the alignment of the retrieved examples with respect to the query c^s, as we will show in the following discussion, it is beneficial to optimize Ψ for better code translation performance.

3.2 Task-Specific Embedding Alignment

Our method involves aligning the Fortran embedding model Ψ using contrastive learning based on CodeBLEU similarity scores, followed by applying this aligned model within a RAG framework for improved cross-language code translation from Fortran to C++, as shown in Figure 1I.

Embedding Similarity: We directly leverage the CodeBLEU similarity computed from the language model G to train an aligned embedding module Ψ for the downstream code translation task. The following discusses how to extract the CodeBLEU similarity from G.

From a source dataset of Fortran code snippets $\mathcal{D}^F = \{c_i^s\}_{i=1}^N$, we generate the corresponding C++ translations $\mathcal{D}^C = \{c_i^t\}_{i=1}^N$ using G without RAG retrieval:

$$c_i^t = G(c_i^s), \quad \forall i = 1, \ldots, N \tag{4}$$

Then, we compute the pairwise CodeBLEU similarity scores (Ren et al., 2020) between all generated translation pairs (c_i^t, c_j^t):

$$S_{ij}^t = \text{CodeBLEU}(c_i^t, c_j^t) \tag{5}$$

where the CodeBLEU score matrix $S^t \in [0,1]^{N\times N}$ is a weighted linear combination of four components: the n-gram match $S_{\text{n-gram}}$, the weighted n-gram match $S_{\text{w-n-gram}}$, the syntactic AST match S_{syntax}, and the semantic data flow match S_{semantic}. These components capture the syntactic and semantic similarities between the generated C++ translations: $S_{\text{n-gram}}$ is the traditional BLEU score up to n-grams, $S_{\text{w-n-gram}}$ assigns weights to n-grams based on their importance, S_{syntax} measures the similarity between the abstract syntax trees (AST) of the code snippets and S_{semantic} assesses the similarity in data flow between code snippets.

Intuitively, a high value of S_{ij}^t indicates that the source code snippets c_i^s and c_j^s, when translated by G, produce similar target code, suggesting that c_i^s and c_j^s are semantically similar with respect to the translation task. Therefore, our approach aims to learn a fine-tuned embedding module Ψ that

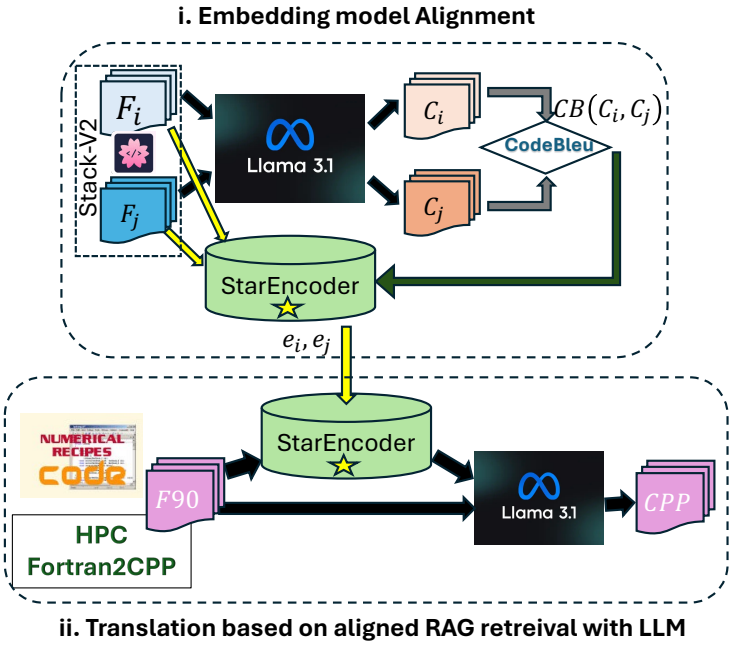

Figure 1: Overview of the proposed pipeline. i) The LLM generates pairwise code translations, which are evaluated using the CodeBLEU metric. ii) The resulting similarity scores are used to guide contrastive learning for semantic alignment of the embedding model.

utilizes S^t_{ij} to enhance code embedding alignment. The approach is expected to guide Ψ in a way that enhances the code translation task leveraging G.

Embedding Alignment: To align the embedding space of code snippets with the semantic similarities measured by CodeBLEU, we propose the Soft Information Noise-Contrastive Estimation (S-InfoNCE) loss applied to the embeddings resulting from the trainable embedding module Ψ. On a high level, our proposed S-InfoNCE can be considered a soft version of the InfoNCE loss proposed for contrastive learning (van den Oord et al., 2018). In the following, we provide the description for the S-InfoNCE loss and Lemma 1 characterizing the stationary condition resulting from the S-InfoNCE. The result helps describing the influence of the loss on the learnt representation.

Given a batch of N code snippets, we compute their embeddings $c^s_{\Psi_i} = \Psi(c^s_i)$ and then calculate the pairwise cosine similarities between those embeddings, scaled by a temperature parameter $\tau > 0$:

$$S^s_{\Psi_{ij}} = \frac{1}{\tau} \frac{c^s_{\Psi_i} \cdot c^s_{\Psi_j}}{\|c^s_{\Psi_i}\| \|c^s_{\Psi_j}\|} \tag{6}$$

Our proposed S-InfoNCE loss integrates these continuous similarity scores to weigh the contribution of each pair. Specifically, the loss component between code i with respect to code j is given as:

$$l^{\text{S-InfoNCE}}_{ij}(\Psi) = -S^t_{ij} \log \left(\frac{\exp(S^s_{\Psi_{ij}})}{\sum_{k=1}^{N} \exp(S^s_{\Psi_{ik}})} \right) \tag{7}$$

and the S-InfoNCE loss is the sum over all code pairs:

$$\mathcal{L}^{\text{S-InfoNCE}}(\Psi) = \sum_{i=1}^{N} \sum_{j=1}^{N} l^{\text{S-InfoNCE}}_{ij}(\Psi) \tag{8}$$

Finally, the embedding Ψ is optimized by minimizing $\mathcal{L}^{\text{S-InfoNCE}}(\Psi)$ using gradient descent.

Compared to the conventional InfoNCE loss for contrastive learning (van den Oord et al., 2018), our proposed loss differs in its usage of S^t_{ij} as a soft indicator for encoding a continuous similarity between the pair (i, j), rather than a binary indicator of class membership (same class or not). This gives rise to the term *soft* InfoNCE, or S-InfoNCE. In the typical InfoNCE loss, the term l_{ij} is included only if the pair (i, j) belongs to the same class, assuming discrete classes are available. However, since such discrete class labels do not exist in the code translation task, we adopt S^t_{ij} as a soft version of this indicator function, allowing for a more nuanced representation of similarity between code pairs. To further elaborate on the impact of S-InfoNCE, we

provide Lemma 1 characterizing its stationary conditions:

Lemma 1. *The stationary points of the S-InfoNCE loss (Equation 8) satisfy:*

$$\frac{\exp(S^s_{\Psi^*_{ij}})}{\sum_{k=1}^N \exp(S^s_{\Psi^*_{ik}})} = \frac{S^t_{ij}}{\sum_{k=1}^N S^t_{ik}}, \tag{9}$$

for all $i, j \in \{1, \ldots, N\}$.

Furthermore, the optimal loss is the weighted sum of the entropy of the CodeBLEU similarity distribution for each input code i:

$$\mathcal{L}^{\text{S-InfoNCE}}(\Psi^*) = \sum_{i=1}^N \left(\sum_{k=1}^N S^t_{ik} \right) H(\boldsymbol{p}^*_i), \tag{10}$$

where H is the entropy function and $\mathbf{p}^*_i$ is a probability vector whose j-th component is

$$p^*_{ij} = \frac{S^t_{ij}}{\sum_{k=1}^N S^t_{ik}}. \tag{11}$$

Proof. For brevity, let us define:

- $\alpha_{ij} = S^t_{ij}$: the CodeBLEU similarity between the target code translations c^t_i and c^t_j.
- $p_{ij}(\Psi) = \exp(S^s_{\Psi_{ij}})/Z_i$, where $Z_i = \sum_{k=1}^N \exp(S^s_{\Psi_{ik}})$: the normalized exponential of the cosine similarity between the embeddings of source code snippets c^s_i and c^s_j.

The S-InfoNCE loss can be rewritten as:

$$\mathcal{L}^{\text{S-InfoNCE}}(\Psi) = -\sum_{i=1}^N \sum_{j=1}^N \alpha_{ij} \log p_{ij}(\Psi). \tag{12}$$

The minimization of $\mathcal{L}^{\text{S-InfoNCE}}(\Psi)$ can be viewed as a constrained optimization problem over the variables $p_{ij}(\Psi)$, subject to the normalization constraints:

$$\sum_{j=1}^N p_{ij}(\Psi) = 1, \quad \forall i \in \{1, \ldots, N\}. \tag{13}$$

Thus, we can formulate the Lagrangian $\mathcal{L}$:

$$\mathcal{L} = -\sum_{i=1}^N \sum_{j=1}^N \alpha_{ij} \log p_{ij}(\Psi) + \sum_{i=1}^N \lambda_i \left(\sum_{j=1}^N p_{ij}(\Psi) - 1 \right). \tag{14}$$

To find the stationary points, we take the derivative of $\mathcal{L}$ with respect to $p_{ij}(\Psi)$ and set it to zero:

$$\frac{\partial \mathcal{L}}{\partial p_{ij}(\Psi)} = -\frac{\alpha_{ij}}{p_{ij}(\Psi)} + \lambda_i = 0. \tag{15}$$

Solving for $p_{ij}(\Psi)$, we get:

$$p_{ij}(\Psi) = \frac{\alpha_{ij}}{\lambda_i}. \tag{16}$$

Applying the normalization constraint gives us:

$$\sum_{j=1}^N \frac{\alpha_{ij}}{\lambda_i} = \sum_{j=1}^N p_{ij}(\Psi) = \sum_{j=1}^N \frac{\alpha_{ij}}{\lambda_i} = 1 \tag{17}$$

$$\Rightarrow \lambda_i = \sum_{j=1}^N \alpha_{ij}. \tag{18}$$

Substituting λ_i back into $p_{ij}(\Psi)$, we obtain the stationary condition:

$$p^*_{ij} = \frac{\alpha_{ij}}{\sum_{k=1}^N \alpha_{ik}} = \frac{S^t_{ij}}{\sum_{k=1}^N S^t_{ik}}. \tag{19}$$

Substituting $p_{ij}(\Psi^*)$ back into the loss function:

$$\mathcal{L}^{\text{S-InfoNCE}}(\Psi^*) = -\sum_{i=1}^N \sum_{k=1}^N \alpha_{ik} \log \left(\frac{\alpha_{ik}}{\sum_{j=1}^N \alpha_{ij}} \right) = \sum_{i=1}^N \left(\sum_{k=1}^N S^t_{ik} \right) H(\boldsymbol{p}^*_i). \tag{20}$$

□

From the lemma, we can see that minimizing the S-InfoNCE loss encourages embeddings of semantically similar code snippets, i.e., those with higher target CodeBLEU score S^t_{ij}, to have higher cosine similarities $S^s_{\Psi_{ij}}$, thereby aligning them closer in the embedding space. The temperature parameter τ controls the concentration of the distribution: a lower τ sharpens the softmax distribution, making the embedding model focus more on the most similar pairs.

Retrieval-Augmented Generation with Aligned Embeddings: After aligning the embedding model Ψ, we integrate it into the RAG framework to enhance the translation process (Figure 1II). In particular, given a query Fortran code snippet c^s, we compute its embedding c^s_Ψ then retrieve the top-k Fortran code snippets $\{c^s_{r_1}, c^s_{r_2}, \ldots, c^s_{r_k}\}$ from the corpus $\mathcal{C}$ by maximizing the cosine similarity between embeddings. The

corresponding C++ translations $\{c^t_{r_1}, c^t_{r_2}, \ldots, c^t_{r_k}\}$ are then retrieved alongside the source code snippets. These retrieved pairs $\{(c^s_{r_j}, c^t_{r_j})\}_{j=1}^k$ are used to augment the input to the language model G, providing additional context:

$$\hat{c}^t = G\left(c^s, \{(c^s_{r_j}, c^t_{r_j})\}_{j=1}^k\right). \tag{21}$$

By incorporating the optimized embedding function Ψ into the RAG setup, we enhance the performance of the language model without the need for fine-tuning. The retrieval mechanism now provides more relevant examples that are closely aligned with the translation task, leading to more accurate and aligned translations as demonstrated in Appendix A.

4 Experiments and Results

In our study, we utilized three datasets to enhance code translation through RAG and embedding alignment. The HPC Fortran2CPP dataset (Lei et al., 2023), comprising 315 Fortran-C++ code pairs, and the Numerical Recipes dataset (Press et al., 1988), containing 298 Fortran-C++ pairs, were employed for RAG retrieval and evaluation with LLMs. Additionally, we used the Stack-V2 dataset (Lozhkov et al., 2024), which includes over 500,000 Fortran code snippets, for RAG alignment. From Stack-V2, we sampled 25,000 high-quality and diverse Fortran code snippets by selecting files larger than 500 bytes and prioritizing those with the highest combined star and fork counts, indicating relevance and popularity. Since Stack-V2 lacks Fortran-C++ pairs, we extracted files containing metadata, code, and comments, and utilized the Llama 3.1-70B Instruct model to extract executable Fortran code, discarding other metadata. We selected the StarCoder model (Li et al., 2023) with 125M parameters as the embedding backbone for our RAG pipeline and aligned it using contrastive learning on the Stack-V2 dataset. Initially, we use the LLaMA 3.1-8B model to translate the cleaned Fortran code snippets into corresponding C++ code. After code translaton, we computed pairwise CodeBLEU scores between the generated C++ code snippets to quantify the syntactic and semantic similarities of their translations. Leveraging these CodeBLEU metrics and the embeddings from the Fortran codes, we employed the proposed Soft-InfoNCE loss function with a temperature of 0.1 to align the embeddings, effectively training the embedding model to map semantically similar code snippets closer in the embedding space.

The embedding model was trained using the Adam optimizer with a learning rate of 10^{-3} and a batch size of 128 per GPU, sampling approximately 1,280,000 code pairs for alignment. This training process was distributed across 256 GH200 GPUs to accelerate the process, though it can also be performed on fewer GPUs at a significantly slower pace. Training on 256 GH200 GPUs took approximately 15 minutes per epoch, with early stopping at epoch 20. This scales to around 60 minutes per epoch with 64 GPUs and 2 hours per epoch with 32 GPUs. While training a RAG model does incur computational overhead, it remains significantly less expensive than fine-tuning a multi-billion-parameter LLM. After alignment, we integrated the embedding model into the RAG pipeline, storing Fortran-C++ pairs along with their Fortran embeddings in a vector database. We then evaluated the performance using the LLaMA 3.1-8B, LLaMA 3.1-70B, Mistral123B, and Mixtral 8x22B models—all instruct-tuned—under zero-shot, 1-shot, 2-shot, and 3-shot settings. The evaluation was conducted on the benchmark datasets HPC Fortran2C++ and Numerical Recipes, following the setup described by (Bhattarai et al., 2024). The CodeBLEU scores for both the aligned and unaligned models were obtained by comparing the RAG-augmented generated C++ translations against the ground truth C++ code.

Figure 2 shows scatter-plots of CodeBLEU scores for code samples produced using RAG retrieval with aligned versus unaligned embeddings derived from StarEncoder. Symbols crosses, pluses and triangles respectively indicate whether the sample was evaluated using a 1-shot, 2-shot, or 3-shot method. The red dashed lines delineates the boundary where the aligned samples have the same CodeBLEU score as the non-aligned ones, and across all four tested datasets, we observed a majority of samples above the red line, indicating that the aligned model produces translated codes closer to ground truth. In other words, the results in Figure 2 demonstrate that aligned embeddings significantly improve translation quality for each Fortran-to-C++ code translation task. Specifically, on the HPC Fortran2C++ dataset, averaged over all shot counts and models, the aligned embeddings achieved an average CodeBLEU score of 0.73, whereas unaligned embeddings achieve 0.64. On the Numerical Recipes dataset, aligned embeddings yielded

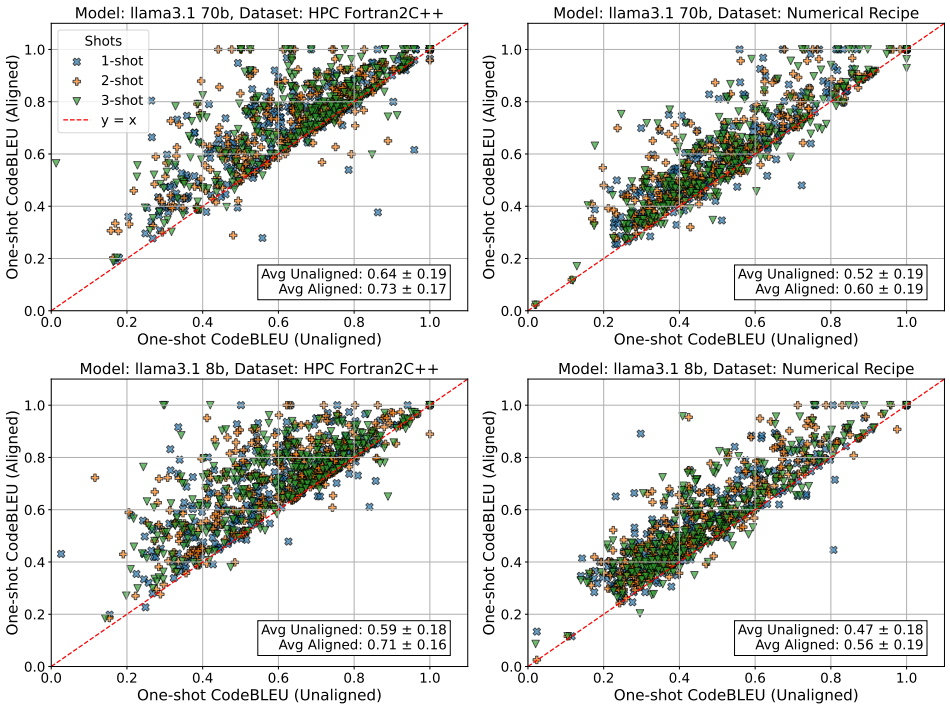

Figure 2: Scatter plots comparing the unaligned and aligned One-shot CodeBLEU scores across different shot counts (1-shot, 2-shot, 3-shot) for two models (llama3.1 70b and llama3.1 8b) and two datasets (Numerical Recipe and HPC Fortran2C++ Dataset). Each point represents a shot count, and the red dashed line represents the reference where the unaligned and aligned scores are equal. The text box in each subplot displays the average CodeBLEU performance and standard deviation for aligned vs. unaligned RAG translation across the few-shot configurations.

Table 1: Delta in Mean CodeBLEU scores between Zero- and Few-Shot prompts. The values are presented as Unaligned/Aligned scores.

		Δ in CodeBLEU scores (Unaligned / Aligned)			
Dataset	**Model**	**Zero-shot**	**1-shot**	**2-shot**	**3-shot**
HPC Fortran2++	llama3.1 70b	0.364	+0.262/+0.346	+0.275/+0.371	+0.281/+0.377
	llama3.1 8b	0.342	+0.237/+0.346	+0.261/+0.376	+0.252/+0.374
	mistral123b	0.367	+0.197/+0.241	+0.210/+0.265	+0.215/+0.271
	mixtral-8x22b	0.376	+0.237/+0.273	+0.261/+0.344	+0.233/+0.304
numerical_receipe	llama3.1 70b	0.280	+0.232/+0.313	+0.243/+0.329	+0.243/+0.317
	llama3.1 8b	0.276	+0.181/+0.268	+0.195/+0.292	+0.201/+0.289
	mistral123b	0.281	+0.138/+0.169	+0.132/+0.183	+0.135/+0.211
	mixtral-8x22b	0.280	+0.200/+0.245	+0.228/+0.296	+0.232/+0.312

an average CodeBLEU score of 0.60, outperforming the unaligned case at 0.52. These substantial improvements highlight the effectiveness of our method in enhancing translation accuracy.

Figure 3 further corroborates these findings by presenting the distribution of CodeBLEU scores across various experimental configurations. The box plots reveal that aligned embeddings not only increase the median scores but also reduce performance variability. This indicates that our approach consistently enhances translation quality and leads to more reliable code translations. The consistent improvements across different model sizes (8B and 70B parameters) and datasets demonstrate the robustness and scalability of our method.

Table 1 presents the mean CodeBLEU scores for zero-shot and few-shot prompting strategies using both unaligned and aligned embedding models across different language models and datasets. A key observation is that the aligned embedding

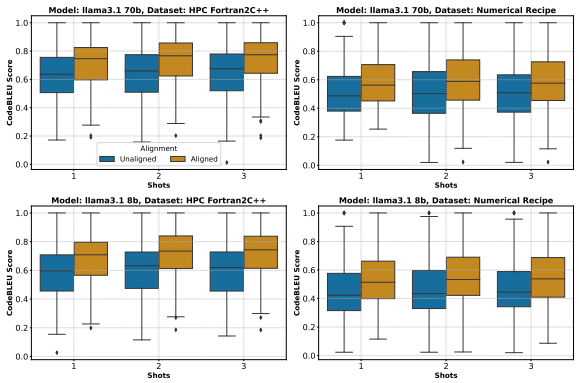

Figure 3: Box plots illustrating the distribution of CodeBLEU scores across various shot counts (1-shot, 2-shot, 3-shot) for both unaligned and aligned models. The results are presented for two models (llama3.1 70b and llama3.1 8b) across two datasets (Numerical Recipe and HPC Fortran2C++ Dataset)

models consistently achieve higher CodeBLEU scores compared to unaligned models when transitioning from zero-shot to few-shot settings. For instance, on the HPC Fortran2C++ dataset with the `LLaMA3.1 70B` model, the aligned model improves from 0.364 to 0.710 (+0.346) in the 1-shot setting, surpassing the unaligned model's improvement from 0.364 to 0.626 (+0.262). Similar trends are observed with the `LLaMA3.1 8B` model, where the aligned model increases from 0.342 to 0.688 (+0.346), compared to the unaligned model's increase from 0.342 to 0.579 (+0.237). The `Mistral 13B` and `Mixtral 8x22B` models also exhibit greater improvements with aligned embeddings in few-shot settings, confirming the benefit of embedding alignment across different architectures. On the Numerical Recipes dataset, the aligned models again demonstrate superior improvements over unaligned models. For example, the `LLaMA3.1 70B` aligned model improves from 0.280 to 0.593 (+0.313) in the 1-shot setting, exceeding the unaligned model's increase from 0.280 to 0.512 (+0.232). This consistent pattern across datasets reinforces the advantage of embedding alignment in enhancing code translation performance. We acknowledge that CodeBLEU may not capture all functional nuances. Therefore, we performed a small-scale manual check (Appendix A) on a subset of translations. While we observed that a majority compiled and produced the expected outputs, further large-scale functional evaluation remains an important future direction.

These results indicate that embedding alignment significantly enhances the models' capacity to exploit few-shot prompts, leading to superior code translation performance as measured by CodeBLEU scores. Alignment optimizes the embedding space to better capture the syntactic and semantic nuances of code translation tasks, thereby augmenting the models' few-shot learning capabilities. Additionally, larger models tend to outperform smaller ones. The `LLaMA3.1 70B` model consistently achieves higher CodeBLEU scores than the `LLaMA3.1 8B` model across both datasets and embedding types. The strong performance of the `Mixtral 8x22B` model, which combines multiple experts, highlights the benefits of increased model capacity. Furthermore, diminishing marginal gains are observed when increasing the number of shots beyond two, suggesting that the majority of performance improvements are realized with just one or two examples. This indicates that while few-shot examples are beneficial, adding more beyond a certain point yields limited additional gains.

5 Conclusion

We introduced a novel method for enhancing cross-language code translation from Fortran to C++ by aligning embeddings within a RAG framework. By leveraging contrastive learning based on CodeBLEU similarity scores, we aligned the Fortran embedding model so that code snippets yielding high-quality translations are positioned closer in the embedding space. This alignment enables the RAG system to retrieve semantically meaningful examples that effectively guide th LLM during code generation. Our experimental results demonstrate substantial improvements in translation quality without the need for fine-tuning the LLM. Specifically, using aligned embeddings increased the average CodeBLEU score from 0.64 to 0.73 on the HPC Fortran2C++ dataset and from 0.52 to 0.60 on the Numerical Recipes dataset, representing relative improvements of approximately 14% and 15%, respectively. The larger model (`llama3.1 70b`) consistently outperformed the smaller model (`llama3.1 8b`), indicating that increased model capacity enhances the effectiveness of our approach. Additionally, we observed diminishing returns beyond two-shot prompting, suggesting that most performance gains are achieved with just one or two examples. Thus, our approach significantly improves code translation performance by optimizing the retrieval mechanism through task-specific em-

bedding alignment, rather than relying on computationally expensive fine-tuning of the LLM. This method is computationally efficient, scalable, and adaptable to other code translation tasks, particularly when aligned datasets are scarce or evaluation metrics like CodeBLEU are critical. Future work could extend this alignment strategy to additional programming languages and explore integrating other evaluation metrics to further enhance translation quality.

6 Limitations

Our approach leverages CodeBLEU as a task-specific metric for performing contrastive learning via a custom Soft-InfoNCE loss in the alignment of embedding models for code translation. While this approach introduces several improvements, it also brings specific limitations. First, using CodeBLEU as the basis for contrastive learning focuses primarily on syntactic and semantic alignment, which may not always translate into functional equivalence. CodeBLEU, while effective at evaluating linguistic features of generated code, does not fully capture the functional behavior of code, meaning that two semantically similar snippets could still behave differently at runtime (Ren et al., 2020). This limitation can lead to cases where the retrieval mechanism selects semantically similar but functionally incorrect examples, impacting the overall quality of the translation task. Second, contrastive learning, particularly with InfoNCE loss, relies heavily on the assumption that maximizing the similarity between pairs (based on CodeBLEU) leads to better downstream performance. However, InfoNCE loss is limited by its focus on pulling positive samples closer while pushing away negative ones, which in the case of code translation, does not always capture the subtle nuances of code equivalence across languages (Khosla et al., 2020). Code snippets with different syntactic structures but similar functionality may be treated as negative examples, leading to a misaligned embedding space and suboptimal retrieval. Third, the granularity of the CodeBLEU score presents an inherent challenge. Since CodeBLEU provides a continuous similarity metric (between 0 and 1), aligning embeddings through InfoNCE loss may not fully capture the wide range of functional similarities or dissimilarities between code snippets. This results in an embedding space that reflects linguistic rather than purely functional similarity, which can lead to errors in retrieval when applied to real-world translation tasks where functional correctness is paramount (Feng et al., 2020). Additionally, the use of CodeBLEU as a basis for contrastive learning is highly dependent on the quality of the generated code samples and their reference translations. Any noise or imperfections in the training data (e.g., low-quality code or inconsistent style) may degrade the alignment process. Since InfoNCE relies on subtle positive and negative distinctions, noisy CodeBLEU scores can introduce ambiguity, further distorting the learning process and leading to poorer retrievals during generation (Wang and Liu, 2021).

7 Acknowledgement

This manuscript has been approved for unlimited release and has been assigned LA-UR-24-33137. 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

Manish Bhattarai, Javier E Santos, Shawn Jones, Ayan Biswas, Boian Alexandrov, and Daniel O'Malley. 2024. Enhancing code translation in language models with few-shot learning via retrieval-augmented generation. *arXiv preprint arXiv:2407.19619*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. *Advances in neural information processing systems*, 33:18661–18673.

Philipp Koehn. 2009. *Statistical machine translation*. Cambridge University Press.

Bin Lei, Caiwen Ding, Le Chen, Pei-Hung Lin, and Chunhua Liao. 2023. 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)*, pages 1–7. IEEE.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

Chuangji Li, Shizhuo Li, and Alan Wang. Retrieval-augmented multi-hop code generation with codellama and unlimiformer.

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

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, et al. 2024. Granite code models: A family of open foundation models for code intelligence. *arXiv preprint arXiv:2405.04324*.

Zach Nussbaum, John X Morris, Brandon Duderstadt, and Andriy Mulyar. 2024. Nomic embed: Training a reproducible long context text embedder. *arXiv preprint arXiv:2402.01613*.

William H Press, William T Vetterling, Saul A Teukolsky, and Brian P Flannery. 1988. *Numerical recipes*. Cambridge University Press, London, England.

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

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*.

Feng Wang and Huaping Liu. 2021. Understanding the behaviour of contrastive loss. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2495–2504.

A Demonstration: Enhancing Fortran-to-C++ Translation Using Aligned RAG on LLaMa3.1 70b Model

A.1 Fortran Code to be Translated

Below is the original Fortran code that needs to be translated into C++:

```fortran
program DRB011_minusminus_orig_yes
    use omp_lib
    implicit none

    integer :: i, len, numNodes, numNodes2
    integer :: x(100)

    len = 100
    numNodes = len
    numNodes2 = 0

    do i = 1, len
        if (MOD(i,2) == 0) then
            x(i) = 5
        else
            x(i) = -5
        end if
    end do

    !$omp parallel do
    do i = numNodes, 1, -1
        if (x(i) <= 0) then
            numNodes2 = numNodes2 - 1
        end if
    end do
    !$omp end parallel do

    print*, "numNodes2 =", numNodes2
end program
```

Listing 1: Fortran Code

A.2 Ground Truth

Below is the human-generated ground truth Fortran-to-C++ translation, which serves as the baseline:

```cpp
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char* argv[]) {
    int i;
    int len = 100;
    int numNodes = len;
    int numNodes2 = 0;
    int x[100];

    // initialize x[]
    for (i = 0; i < len; i++) {
        if (i % 2 == 0)
            x[i] = 5;
        else
            x[i] = -5;
    }

#pragma omp parallel for
    for (i = numNodes - 1; i > -1; --i)
    {
        if (x[i] <= 0) {
            numNodes2--;
```

```
        }
    }

    printf("numNodes2 = %d\n", numNodes2
    );
    return 0;
}
```

Listing 2: Ground Truth

A.3 Zero-shot Translation

The zero-shot output from the LLM demonstrates basic translation abilities but with structural issues:

```
#include <iostream>
#include <omp.h>

int main() {
    const int len = 100;
    int numNodes = len;
    int numNodes2 = 0;
    int x[len];

    for (int i = 1; i <= len; i++) {
        if (i % 2 == 0) {
            x[i - 1] = 5;
        } else {
            x[i - 1] = -5;
        }
    }

    #pragma omp parallel for reduction
    (-:numNodes2)
    for (int i = numNodes; i >= 1; i--)
    {
        if (x[i - 1] <= 0) {
            numNodes2--;
        }
    }

    std::cout << "numNodes2 = " <<
    numNodes2 << std::endl;

    return 0;
}
```

Listing 3: Zero-shot Translation

A.4 Unaligned RAG One-shot Translation

The one-shot output improves the translation, but some structural issues still remain. Specifically, the initialization of variables and handling of array indices are slightly different:

```
#include <stdio.h>

int main() {
    int i, len, numNodes, numNodes2;
    int x[100];

    len = 100;
    numNodes = len;
    numNodes2 = 0;

    for (i = 0; i < len; i++) {
        if ((i + 1) % 2 == 0) {
            x[i] = 5;
        } else {
            x[i] = -5;
        }
    }

#pragma omp parallel for reduction(-:
    numNodes2)
    for (i = numNodes - 1; i >= 0; i--)
    {
        if (x[i] <= 0) {
            numNodes2--;
        }
    }

    printf("numNodes2 = %d\n", numNodes2
    );

    return 0;
}
```

Listing 4: One-shot Translation

A.5 Aligned RAG Translation

By aligning the retrieval process to the context of the task, the translation becomes much more accurate, closely matching the ground truth:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]) {
    int i;
    int len = 100;
    int numNodes = len;
    int numNodes2 = 0;
    int x[100];

    for (i = 0; i < len; i++) {
        if ((i % 2) == 0) {
            x[i] = 5;
        } else {
            x[i] = -5;
        }
    }

#pragma omp parallel for
    for (i = numNodes - 1; i >= 0; i--)
    {
        if (x[i] <= 0) {
            numNodes2--;
        }
    }

    printf("numNodes2 = %d\n", numNodes2
    );
    return 0;
}
```

Listing 5: Aligned RAG-based Translation