

MojoBench: Language Modeling and Benchmarks for Mojo

Nishat Raihan¹, Joanna C. S. Santos², Marcos Zampieri¹

¹George Mason University, USA

²University of Notre Dame, USA

mraihan2@gmu.edu

Abstract

The recently introduced Mojo¹ programming language (PL) by Modular, has received significant attention in the scientific community due to its claimed significant speed boost over Python. Despite advancements in code Large Language Models (LLMs) across various PLs, Mojo remains unexplored in this context. To address this gap, we introduce MojoBench, the first framework for Mojo code generation. MojoBench includes HumanEval-Mojo, a benchmark dataset designed for evaluating code LLMs on Mojo, and Mojo-Coder, the first LLM pretrained and finetuned for Mojo code generation, which supports instructions in 5 natural languages (NLs). Our results show that Mojo-Coder achieves a 30-35% performance improvement over leading models like GPT-4o and Claude-3.5-Sonnet. Furthermore, we provide insights into LLM behavior with underrepresented and unseen PLs, offering potential strategies for enhancing model adaptability. MojoBench contributes to our understanding of LLM capabilities and limitations in emerging programming paradigms fostering more robust code generation systems.

1 Introduction

LLMs demonstrate exceptional capabilities in both NLP and coding tasks, including generating executable code snippets from NL descriptions. While general-purpose models like GPT-4 (OpenAI, 2024) and the LLaMA family (Touvron et al., 2023a,b; Dubey et al., 2024) exhibit strong coding abilities, task-specific models such as CodeLLaMA (Roziere et al., 2023) and WizardCoder (Luo et al., 2023), fine-tuned for code generation, often outperform them despite smaller model sizes. More recent models like CodeGemma (CodeGemma Team, 2024) and CodeStral (Mistral AI, 2024) have expanded support to multiple

PLs, moving beyond the predominantly Python-focused earlier models. Similar trends are also seen in existing code generation benchmarks, with limited focus on non-Python PLs.

We argue that the disproportionate focus on Python and a few other mainstream PLs overlooks the critical need to create resources for emerging and more specialized PLs. Some of these PLs, despite limited online presence, often play major roles in some sectors and domains. Mojo exemplifies this phenomenon, having rapidly ascended to the Top 100 most utilized PLs and capturing significant interest in the Machine Learning (ML) community.² Current LLMs fail to support Mojo and similar emerging PLs, even for basic tasks like code completion. For example, as illustrated in Listing 1, Clause 3.5 Sonnet generates *Python* code instead of *Mojo* code, as explicitly instructed. This glaring disparity demands immediate attention and underscores the urgent need for more inclusive, diverse PL support in LLM development.

```
1 # A Mojo function to reverse a string
2
3 def reverse_string(input_string):
4     return input_string[::-1]
5
6 reverse_string("hello")
```

Listing 1: A Python code snippet, generated by *Claude-3.5-Sonnet* when prompted to “write a function in Mojo to reverse a string”.

To address this need, we introduce MojoBench, a comprehensive framework for Mojo code *evaluation* and *generation*. Within this framework, we present the HumanEval-Mojo benchmark, designed to assess model performance on Mojo, a PL previously excluded from such evaluations. This benchmark allows us to examine state-of-the-art (SOTA) LLM performance on a PL largely unseen

¹modular.com/mojo/

²tiobe.com/tiobe-index/

in training data. Mojo, having been introduced recently (2023) with a limited web presence, serves as an ideal candidate for this investigation.

As a part of MojoBench, we also develop Mojo-Coder, a specialized family of Code LLMs trained for Mojo code generation from NL instructions. Our evaluations demonstrate that Mojo-Coder outperforms SOTA models such as GPT-4o (OpenAI, 2024) and CodeLLaMA (Roziere et al., 2023). Through multilingual supervised fine-tuning (SFT), Mojo-Coder supports instructions in five NLs: English, German, French, Spanish, and Bangla.

Our main contributions with MojoBench are:

- HumanEval-Mojo, the first benchmark designed specifically for evaluating Code LLMs on Mojo programming tasks.
- Mojo-Coder, a novel family of Code LLMs, pretrained and finetuned to surpass SOTA models in Mojo code generation, with support for five natural languages.
- The open-source release of the model, benchmark, two instruction-tuning datasets (Mojo-SFT and multilingual Mojo-mSFT), and the Mojo-Corpus to facilitate further research and development³.

We use the suite of resources in MojoBench to address two important research questions (RQs):

- **RQ1:** How do LLMs perform on coding tasks in PLs that are either unseen or minimally represented in their training data, and what challenges are encountered?
- **RQ2:** How can LLMs be effectively adapted to emerging or underrepresented PLs with limited resources?

2 Background & Motivation

Mojo, introduced by Modular in 2023, is engineered for high-performance computing and machine learning. With its static typing, manual memory management, and SIMD (Single Instruction, Multiple Data) support, Mojo compiles directly to machine code, rivaling C++ performance and outpacing Python by up to 68,000 times according to Deo et al. (2024). At its core, the MLIR (Multi-Level Intermediate Representation)

compiler framework enables advanced optimizations crucial for AI applications⁴.

A critical limitation of existing LLMs is the lack of support for emerging PLs like Mojo. Most LLMs primarily serve established languages such as Python, but this oversight hinders Mojo’s adoption and optimization in AI-intensive applications where its performance edge is most needed. Leading models including GPT-4 (OpenAI, 2024), CodeLLaMA (Roziere et al., 2023), and WizardCoder (Luo et al., 2023) still lack any Mojo support, despite its impressive capabilities. This deficiency limits developers’ access to AI-driven tools for enhancing productivity and optimizing Mojo implementations. A dedicated Mojo-compatible LLM as presented in MojoBench will help to bridge this gap and equip developers with powerful tools.

3 Related Work

While there has been no specific work on Mojo, related work has explored PL-specific models and benchmarks.

PL-specific Corpora Code LLMs typically train on a combination of large NL corpora like Common Crawl (Patel and Patel, 2020) and multi-PL corpora such as The Stack (Kocetkov et al., 2022; Lozhkov et al., 2024) and CodeParrot (Tunstall et al., 2022). Although PL-specific corpora are rare, they can be extracted from larger datasets, albeit through a challenging process. However, using multi-PL corpora for specific PL tasks often leads to suboptimal performance, as demonstrated by the CodeGemma Team.

PL-specific Benchmarks Well-established benchmarks like HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and CSEPrompts (Raihan et al., 2024b) primarily evaluate Python code generation. Recent extensions such as HumanEval-XL (Peng et al., 2024) and mHumanEval (Raihan et al., 2024a) have expanded language coverage to include Java, C, C++, and others. Despite these advancements, many important PLs from the over 100 languages with substantial user bases remain underrepresented.

PL-specific LLMs Most of Code LLMs, such as MagiCoder (Wei et al., 2023), are tailored for Python, being fine-tuned primarily to generate

³github.com/mraihan-gmu/MojoBench

⁴github.com/tairov/llama2.mojo

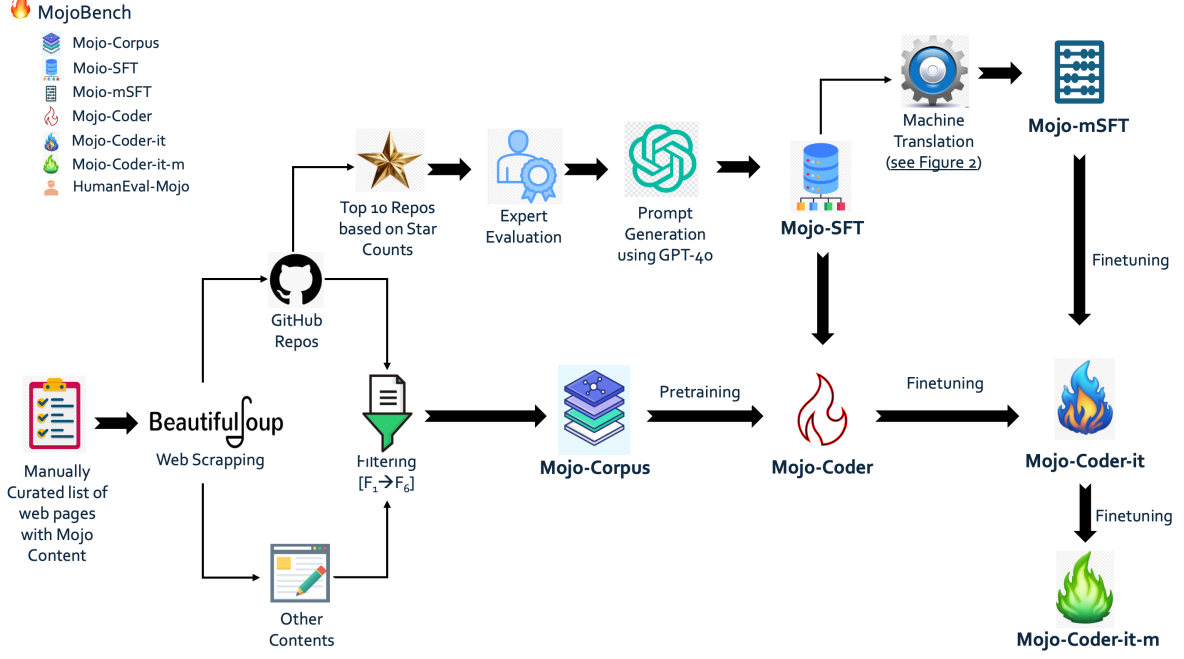


Figure 1: The complete workflow of developing MojoBench and all of its components. See Section 4.3 for HumanEval-Mojo.

Python code. PL-specific models like StarCoder-Java (Rathinasamy et al., 2024) are exceptions. In contrast, multi-PL models are more widespread, with proprietary systems like GPT-4 (OpenAI, 2024) and the Claude-family standing out as prominent examples.

4 MojoBench

This section provides an overview of our MojoBench framework and its components. Figure 1 shows its complete workflow.

4.1 Mojo-Corpus: A Corpus of Mojo Code

MojoBench includes a task-specific corpus. We curate data from publicly available sources, including Modular’s official documentation⁵, GitHub repositories, blogs, tutorials, and news articles. As discussed in the Ethical Considerations Section, we collect data from open repositories - comprehensive list of these sources is in Appendix A. We use BeautifulSoup⁶, a commonly used Python library for data collection.

Following data collection, we implement a comprehensive cleaning process to ensure corpus quality. This process applies six sequential filters (F_1 through F_6) to the initial Mojo corpus (MC),

which contains 79,368,439 tokens. Table 1 displays the token count after each filtering stage. More details about these filters are provided in Appendix B.

Filter Description	# Tokens
None All Collected Contents	79,368,439
F_1 Removes non-Apache 2.0 licensed samples.	42,245,342
F_2 Removes Python-specific code snippets.	21,973,419
F_3 Ensures samples have at least 3 meaningful paragraphs.	15,410,741
F_4 Removes samples with excessive internal repetition.	10,112,466
F_5 Removes duplicate samples across the corpus.	7,145,333
F_6 Filters non-English content using fast-Text.	6,583,948

Table 1: Filtering all collected tokens following specific filtering criteria. For convenience, each filter is denoted by F_i . The right-most column shows the token count after applying each filter.

4.2 Mojo-SFT & Mojo-mSFT

MojoBench also includes two separate instruction datasets. Mojo-SFT, comprising instructions exclusively in English, and Mojo-mSFT, encompassing instructions in Spanish, German, French, and Bangla.

⁵docs.modular.com/

⁶beautiful-soup-4.readthedocs.io/en

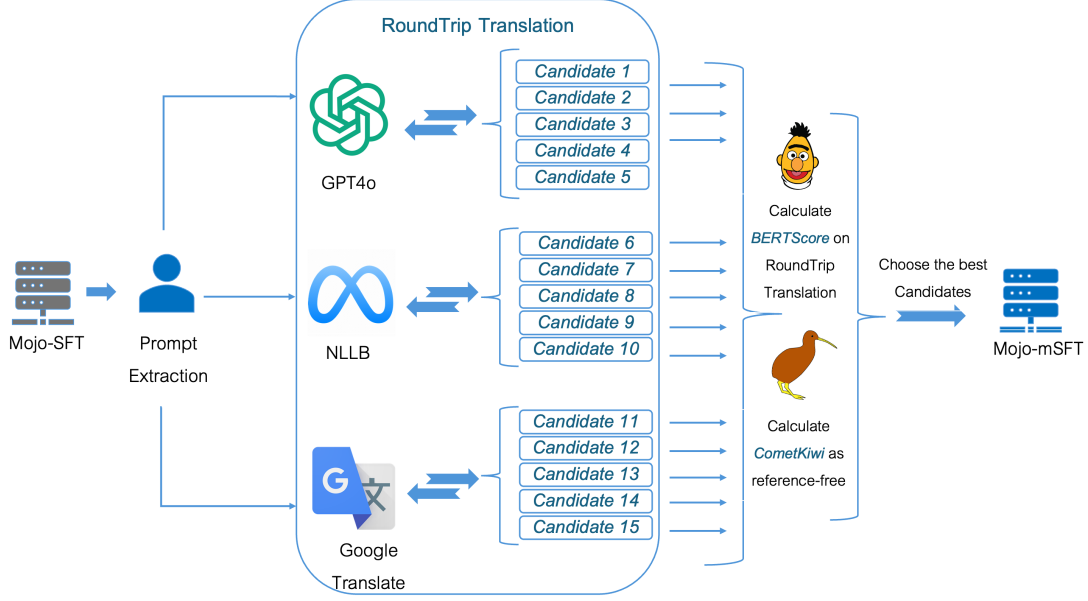


Figure 2: The workflow of compiling Mojo-mSFT from Mojo-SFT. Similar to the approach adopted by (Raihan et al., 2024a).

4.2.1 Mojo-SFT

We initiate by aggregating GitHub repositories for the Mojo-Corpus, MC^* . Let $\mathcal{R} = \{r_1, r_2, \dots, r_n\}$ be the repositories licensed under Apache 2.0 that have undergone all transitions (T_1 to T_6) as detailed in Section 4.1. Repositories are ranked by GitHub star count, $s(r_i)$, and the top 10 repositories, $R = \{r_1, r_2, \dots, r_{10}\}$, are selected such that:

$$\forall r_i \in R, \forall r_j \in \mathcal{R} \setminus R : s(r_i) \geq s(r_j)$$

From these, we extract `.mojo` and `.🔥` files, yielding 968 code files. To maintain consistency, we filter files by token count $\tau(f)$ as follows:

$$\tau(f) = \begin{cases} 1 & \text{if } 5 \leq \text{token_count}(f) \leq 500 \\ 0 & \text{otherwise} \end{cases}$$

where f denotes a code file. Expert programmers review these, ultimately selecting 800 high-quality samples, discarding any that are buggy or too lengthy.

For prompt generation, we apply an *AI-in-the-loop* approach using GPT-4o (OpenAI, 2024) via the OpenAI API, producing three paraphrases per prompt. This results in 3,200 prompt-code pairs, with iterative expert refinement yielding four distinct prompts per snippet. Table 2 provides summary statistics for the Mojo-SFT dataset.

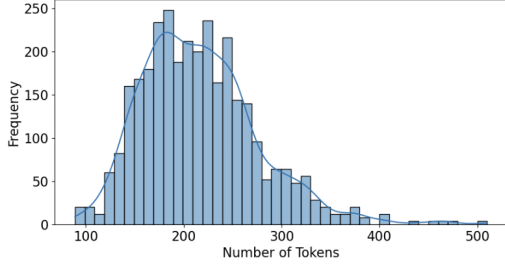
Feature	Statistics
Code blocks	3200
Avg (Tokens)	216.53
Median (Tokens)	210.00
Std. Dev. (Tokens)	59.77
Range (Tokens)	89 to 510
Comments (Full-line)	16,598
Comments (Inline)	1,480
Definitions (Function)	6,368
Definition (Struct)	12

Table 2: Mojo-SFT - Dataset Card.

4.2.2 Mojo-mSFT

For the multilingual dataset Mojo-mSFT, we utilize a synthetic approach across five languages, following (Raihan et al., 2024a). We employ three machine translation strategies: OpenAI’s GPT-4o (OpenAI, 2024) via API, MetaAI’s NLLB (Costa-jussà et al., 2022), and Google Translate. Figure 2 illustrates the workflow.

To assess translation quality, we apply a dual-metric approach. BERTScore (Zhang et al., 2019) measures similarity using contextual embeddings, while CometKiwi (Rei et al., 2023) evaluates based on human-judgment-trained metrics, capturing both technical accuracy and linguistic naturalness. These metrics, discussed further in Appendix C, provide a comprehensive evaluation of translation quality. We choose the best candidate translation from a set of 15 for each prompt.



Listing 2: Code Lengths in both Mojo-SFT & Mojo-mSFT.

4.3 HumanEval-Mojo

The original HumanEval benchmark (Chen et al., 2021) includes 164 coding tasks, paired with test cases, and initially supported only Python. While support for a total of 43 other PLs have been provided with the works of Yan et al. (2023), Raihan et al. (2024a) and Peng et al. (2024); none of them include Mojo along with a quite a few other widely used PLs.

```
1 def is_prime(n: int):
2     """
3     Return True if the input number n is
4     prime, else return False.
5     A prime number is a number greater
6     than 1 and has no divisors other than
7     1 and itself.
8     """
```

Listing 3: A sample prompt from HumanEval.

We extend the original HumanEval for each of the 164 prompts to construct HumanEval-Mojo, an evaluation benchmark under MojoBench. A sample prompt is shown in Listing 3. We extract the docstrings manually and two human experts convert each of the Python function declarations to Mojo syntax, paired with the extracted docstrings (see Listing 4). To ensure correctness, they are tested on both the local environment and the official Mojo Playground⁷.

```
1 fn sum_squares(n: Int) -> Int:
2     """
3     Given an integer n, return the sum
4     of the squares of all integers from
5     1 to n (inclusive).
6     """
```

Listing 4: A sample prompt from HumanEval-Mojo.

Finally, we provide solutions for each prompt in Mojo; hand-written by two experienced Mojo programmers. While we can not make sure that each

⁷<https://docs.modular.com/mojo/playground>

solution is the most optimal one for the task, we do make sure that they pass all the test cases. One such solution is shown in Listing 5.

```
1 fn count_vowels(s: String) -> Int:
2     """
3     Given a string s, return the number
4     of vowels (a, e, i, o, u) in the
5     string.
6     """
7     let vowels = "aeiouAEIOU"
8     var count: Int = 0
9     for char in s:
10         if char in vowels:
11             count += 1
12     return count
```

Listing 5: A sample hand-crafted solution from HumanEval-Mojo.

Unlike Yan et al. (2023), Peng et al. (2024) or other benchmarks that cover more PLs, but follow an automated pipeline or use machine-generated contents, which are often prone to oversights, ours follow the original workflow, incorporating manual approach with a human-expert in the loop, ensuring better quality and more authenticity.

Annotator Details HumanEval-Mojo involves four expert volunteers. Two of these volunteers handle the conversion of the original Python prompts into Mojo prompts, while the other two focus on crafting canonical Mojo solutions for each prompt. All four volunteers have a background in computer science, possess extensive coding experience, and are particularly skilled in the Mojo programming language.

4.4 Mojo-Coder

4.4.1 Base Model Selection

Since our goal is to train a Code LLM that will be generating output in a new PL and should be able to support instructions in multiple NLs, we start by considering 3 candidates; including code-finetuned models like CodeL-LaMA (Roziere et al., 2023) and CodeGemma (CodeGemma Team, 2024), both trained on multiple PLs and NLs and the recent version of Mistral (v0.3), proficient on multiple NLs with moderate performance on non-Python PLs. Due to the limited training content, we consider the 7B variants for each.

4.4.2 Pretraining

Following transformations F_1 through F_6 , we utilize the refined Mojo corpus MC^* for additional pre-training of our selected base models.

Let θ_0 represent the initial parameters of a base model. We aim to find θ^* as follows:

$$\theta^* = \arg \min_{\theta} \mathcal{L}(\theta; MC^*) \quad (1)$$

where $\mathcal{L}$ is the language modeling loss over the cleaned Mojo corpus MC^* .

Pre-training Setup For the pretraining phase, we utilize a cluster of 8 NVIDIA A100 GPUs, each with 40 GB memory, provided by Lambda Labs⁸. This configuration yields a total of 320 GB GPU memory, complemented by 512 GB system memory and 2 TB disk space. The pretraining process spans approximately 32 hours on this high-performance cluster.

Our custom training pipeline, based on the Hugging Face Transformers framework (Wolf, 2019), includes:

$$\mathcal{D} = \arg \min_{\mathcal{D}} \mathcal{L}(\mathcal{D}; MC^*) \quad (2)$$

$$\tau = \arg \min_{\tau} \mathcal{L}(\tau; \mathcal{T}(M_i)) \quad (3)$$

$$\mathcal{M}(\theta) = \arg \min_{\theta} \mathcal{L}(\theta; \mathcal{F}(M_i)) \quad (4)$$

Here, $\mathcal{D}$ denotes the custom dataset from MC^* , τ represents the tokenizer $\mathcal{T}$, and $\mathcal{M}(\theta)$ is the pre-trained model $\mathcal{F}$ using bfloat16 precision. Both $\mathcal{T}$ and $\mathcal{F}$ derive from model family M_i , ensuring tokenizer-model compatibility. Training hyperparameters are detailed in Appendix ??.

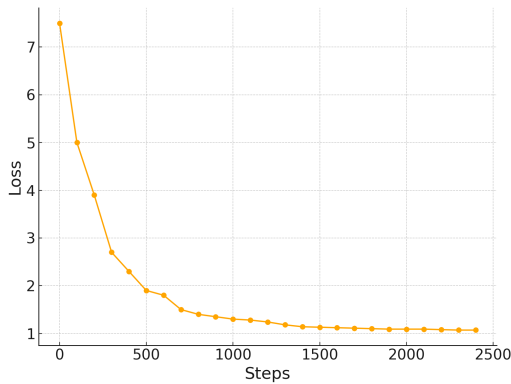


Figure 3: Step vs. Loss graph during pretraining on Mojo-Corpus.

⁸<https://lambdalabs.com/>

Training Architecture The effective batch size B_e is calculated as:

$$B_e = B_d \times G_a \times N_d \quad (5)$$

where $B_d = 32$ (per-device batch size), $G_a = 8$ (gradient accumulation steps), and N_d (number of GPUs). The training steps per epoch S_e are given by:

$$S_e = \left\lceil \frac{|MC^*|}{B_e} \right\rceil \quad (6)$$

where $|MC^*|$ is the total sample count in the Mojo corpus.

Monitoring and Evaluation We log the training loss $\mathcal{L}_s$ every 100 steps via a custom callback function $C(s, \mathcal{L}_s)$. Evaluation occurs every 250 steps on a 10% subset, $MC_{eval}^* \subset MC^*$.

Our checkpointing strategy $\Psi(s, \theta_s, \mathcal{L}_s)$ is:

$$\Psi(s, \theta_s, \mathcal{L}_s) = \begin{cases} \text{save}(\theta_s), & \text{if } s \bmod 250 = 0 \\ \text{save}(\theta_s), & \text{if } \mathcal{L}_s = \min(\mathcal{L}_1, \dots, \mathcal{L}_s) \\ \emptyset, & \text{otherwise} \end{cases} \quad (7)$$

Checkpoints are saved every 250 steps, retaining the model with the lowest evaluation loss. Figure 3 illustrates the decreasing loss over time.

4.4.3 Finetuning

Finetuning Setup The finetuning stage is conducted on a single NVIDIA A100 GPU with 40 GB memory, accessed through Google Colab⁹. This setup is augmented with 80 GB system memory and 256 GB disk space. The finetuning process requires approximately 9 hours to complete on this configuration. This more modest setup is sufficient for the task-specific adaptation of our pretrained models.

We finetune our three pretrained models using Mojo-SFT (English-only) and Mojo-mSFT (multilingual) instruction datasets. Implementing Low-Rank Adaptation (LoRA) (Hu et al., 2021), a Parameter Efficient Finetuning (PEFT) technique (Xu et al., 2023), which optimizes memory usage while preserving performance. We choose LoRA for its ability to significantly reduce the number of trainable parameters, enabling efficient finetuning of large language models without compromising their performance (Hu et al., 2021). For each

⁹<https://colab.research.google.com/>

model M_i ($i \in 1, 2, 3$), we define:

$$\begin{aligned} F_E(M_i) &= \text{LoRA}(M_i, \text{Mojo} - \text{SFT}, r = 64) \\ F_M(M_i) &= \text{LoRA}(M_i, \text{Mojo} - m\text{SFT}, r = 64) \end{aligned} \quad (8)$$

where F_E and F_M are English-only and multilingual finetuning functions, respectively. The LoRA rank $r = 64$ balances parameter efficiency and model capacity.

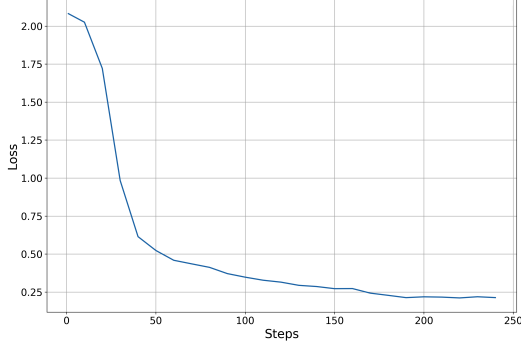


Figure 4: Step vs. Loss graph during finetuning on Mojo-SFT.

Using Flash Attention (Kundu et al., 2024), we set key parameters: 2048 token maximum sequence length, batch size of 8, 4 gradient accumulation steps, and 3 epochs. Learning rate (5×10^{-5}), weight decay (0.02), and 10% warm-up steps ensure stable convergence. Total training steps T are calculated as:

$$T = \left\lfloor \frac{N}{B \times G} \right\rfloor \times E \quad (9)$$

where N is training samples, B is batch size, G is gradient accumulation steps, and E is epochs. Evaluation and model saving occur every 50 steps, with the best model selected by lowest evaluation loss. A cosine learning rate scheduler and 8-bit Adam optimizer (Kingma, 2014) further enhance efficiency without compromising performance. Table 9 in Appendix D lists complete hyperparameters. Figure 4 demonstrates consistent loss decrease during finetuning.

5 Results and Analysis

5.1 Evaluation

Selecting Optimal Models After pretraining and finetuning, we evaluate nine models: three pretrained on Mojo-Corpus, three finetuned on Mojo-SFT (English-only), and three on Mojo-mSFT (multilingual). Using HumanEval-Mojo, we select the best-performing model from each category. Table 3 compares their performance.

We observe poor performance in all three base models. Performance improves significantly after pretraining on Mojo-Corpus and further with Mojo-SFT finetuning. However, Mojo-mSFT finetuning leads to a slight performance decline.

Candidates	HumanEval-Mojo pass@1
CodeGemma	5.1%
CodeLLaMA	4.7%
Mistral	1.3%
CodeGemma-pt	36.7%
CodeLLaMA-pt	34.4%
Mistral-pt	23.1%
CodeGemma-ft	66.4%
CodeLLaMA-ft	54.2%
Mistral-ft	26.9%
CodeGemma-mft	61.5%
CodeLLaMA-mft	49.6%
Mistral-mft	17.3%

Table 3: Candidate Model Selection after pretraining and finetuning. *pt*, *ft*, and *mft* stand for 'pretrained' and 'finetuned,' respectively. (*Pass@1*) is chosen as the accuracy metric. The darker the shade, the better the performance.

Mojo-Coder family We release the three best performer models as the members of the Mojo-Coder family. One pretrained model, Mojo-Coder (*CodeGemma-pt*), allowing practitioners to further develop models with their own instruction datasets and two finetuned models- one as English-only, Mojo-Coder-it (*CodeGemma-ft*) and one multilingual Mojo-Coder-it-m (*CodeGemma-mft*); both SOTA models for Mojo code generation. Appendix E illustrates some code snippets generated by Mojo-Coder.

Comparison We evaluate the Mojo-Coder family of models on HumanEval-Mojo using the *pass@1* metric. The results demonstrate (Table 4) that our models significantly outperform existing state-of-the-art (SOTA) models in this benchmark.

5.2 Insights and Observations

In this section, we describe some key insights obtained from the experiments and revisit our research questions.

Code LLMs vs New PLs Existing code LLMs struggle to generate Mojo code. They often generate non-executable and buggy code snippets. Some examples are given in Appendix F. We also include responses from other SOTA models like

Models	Type	Param	Pass@1
Mistral	Open	7B	1.3%
CodeLLaMA	Open	7B	4.7%
CodeGemma	Open	7B	5.1%
MagiCoder	Open	7B	7.3%
WizardCoder	Open	34B	9.2%
Codestral	Open	23B	9.2%
Code-Qwen	Open	7B	9.9%
DeepSeek-Coder	Open	33B	10.2%
GPT-4o	Close	–	25.5%
Mojo-Coder	Open	7B	36.7%
Claude-3.5-Sonnet	Close	–	39.8%
Mojo-Coder-it-m	Open	7B	61.5%
Mojo-Coder-it	Open	7B	66.4%

Table 4: Performance evaluation of Mojo-Coder family and other Code LLMs on HumanEval-Mojo using *Pass@1*.

GPT-4o (OpenAI, 2024), Claude-3.5-Sonnet, and WizardCoder (Luo et al., 2023). From these results, we gather a key observation to answer our first research question:

RQ₁ How do LLMs perform on coding tasks in PLs that are either unseen or minimally represented in their training data, and what challenges are encountered?

When prompted to write code in Mojo, the models often get the algorithm right for the tasks but the syntax wrong. This is likely due to their exposure to lots of PLs but just not Mojo.

Moreover, when the prompt is in any non-English language (i.e., French), these models underperform even more with mostly unexplainable and poor-quality code snippets (Appendix F).

Language Modeling for Emerging Programming Languages This work extends beyond the finetuning approaches employed by (Wei et al., 2023) and WizardCoder (Luo et al., 2023) by incorporating an additional pretraining phase utilizing a corpus of 6 million tokens. While this corpus is substantially smaller than established programming language datasets (Lozhkov et al., 2024; Tunstall et al., 2022), our results demonstrate significant performance improvements. This outcome underscores the critical importance of acquiring domain-specific knowledge, even from limited data sources. Furthermore, our finetuning dataset, comprising only 3,200 instructions— in contrast to WizardCoder’s 77,000 — accentuates the necessity of targeted, domain-specific learning. These findings directly address our second research question:

RQ₂ How can LLMs be effectively adapted to emerging or underrepresented PLs with limited resources?

LLMs can be effectively adapted for new or underrepresented PLs through domain-specific pretraining corpora (even a small one) and targeted instruction finetuning, prioritizing data quality over quantity to quickly capture language-specific features.

5.3 Ablation Study

For analysis purposes, we also experiment with different sizes of corpora and instruction datasets with all three models from the Mojo-Coder family, and the results further support our approaches and gathered insights.

	0	1M	2M	3M	4M	5M	6M
0	5.1*	12.3	14.8	16.1	21.9	30.1	36.7**
500	15.3	15.6	15.3	17.9	32.1	35.5	38.8
1000	19.6	21.3	20.1	25.7	37.7	39.3	41.6
1500	20.3	27.8	25.3	35.6	41.2	46.7	44.5
2000	22.7	34.9	36.7	38.5	42.3	51.4	53.7
2500	33.3	37.2	39.8	43.1	53.4	59.4	60.1
3000	42.1	43.9	49.1	56.7	55.1	60.2	64.9
3200	42.3	45.3	51.2	53.4	57.9	65.1	66.4***

Table 5: Pretrained and/or finetuned on varied number of tokens (*top-most row*) and/or instructions (*left-most column*). The values represent the model’s *Pass@1* scores. Here, * denotes CodeGemma, ** denotes Mojo-Coder, and *** denotes Mojo-Coder-it.

We derive several hypotheses from the results presented in Table 5:

- The performance of the model exhibits a positive correlation with both the duration of pretraining and the extent of finetuning.
- A model subjected *only* to finetuning can potentially outperform a model that has *only* undergone pretraining, suggesting the critical importance of task-specific adaptation.
- The synergistic application of both pretraining and finetuning appears to be crucial for achieving optimal performance, indicating a complementary relationship between general knowledge acquisition and task-specific optimization.

It is important to note, however, that these three conclusions are drawn from a specific experimental context. Further empirical investigation across diverse datasets and model architectures would be necessary to establish the generalizability of

these findings. We present these observations as promising directions for future research rather than definitive claims.

6 Conclusion

In this work, we presented MojoBench. While most attention have been given to Python and other popular languages, with MojoBench we emphasize the importance of supporting underrepresented PLs, which are often newly developed or with limited online presence, fulfilling specific roles in niche and specialized domains. We focused our work on one such PL - Mojo, which, despite being recently introduced, has been gaining popularity. The paper further contributes by addressing two important RQs on LLMs and code generation.

MojoBench bridges this important gap through the development of the Mojo-Coder family of models, the creation of the HumanEval-Mojo benchmark, and the compilation of two instruction datasets: Mojo-SFT and Mojo-mSFT. Our models demonstrated superior performance, even when compared to much larger proprietary models. Furthermore, we explored potential methods for adapting small Code LLMs to such emerging PLs. We expect that our methodologies and insights would encourage the research community to devote more attention to PLs that are often overlooked but nonetheless maintain a substantial user base.

Limitations

This study introduces MojoBench, the first benchmark for Mojo. It includes multiple datasets, benchmarks, and models. While we prioritized quality over quantity for Mojo-Corpus, Mojo-SFT, and Mojo-mSFT, the limited availability of Mojo content on the web significantly constrained our dataset sizes. As discussed in Section 6, in this work, only open source publicly available data was used which, in turn, reduced the datasets' scope and size. This limitation directly impacts the diversity and richness of the data used for training and evaluation, potentially affecting the models' generalization capabilities.

Furthermore, our models are confined to the $\sim 7\text{B}$ parameter range, a decision influenced by two primary factors. First, the relatively modest size of our datasets aligns better with smaller models, as larger models might overfit on limited data.

Second, the computational intensity of pretraining favors more manageable model sizes, allowing us to iterate and experiment within our resource constraints. While this approach may limit the models' capacity compared to larger counterparts, we believe it strikes a balance between resource efficiency and model performance.

Ethical Considerations

The datasets and models introduced in this paper, including Mojo-Corpus, Mojo-SFT, Mojo-mSFT, and the resulting language models, strictly adhere to the [ACL Ethics Policy](#). We have prioritized ethical data collection practices, using only publicly available sources and respecting intellectual property rights. To ensure safety and reliability, we strongly recommend executing code generated using prompts from HumanEval-Mojo or our Mojo-Coder models in a contained virtual environment. This precaution helps prevent potential issues related to infinite execution loops, memory management, system crashes, and excessive resource consumption. We believe this approach allows researchers and practitioners to maintain a secure and controlled testing environment while confidently exploring and innovating with our resources. We remain committed to ongoing ethical evaluation and welcome community feedback to address any unforeseen concerns.

References

- Jacob Austin, Augustus Odena, Maxwell Nye, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
- Piotr Bojanowski, Édouard Grave, Armand Joulin, et al. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*.
- Mark Chen, Jerry Tworek, , et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Google CodeGemma Team. 2024. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*.
- Marta R Costa-jussà, James Cross, et al. 2022. No language left behind: Scaling human-centered machine translation. *arXiv preprint arXiv:2207.04672*.
- Anuj Kumar Aditya Deo, Swayam Gupta, Roumo Kundu, Piyush Jaiswal, Taha Fatma, and Mohan Kumar Dehury. 2024. Performance and metrics analysis between python3 via mojo. In *2024 2nd Inter-*

- national Conference on Sustainable Computing and Smart Systems (ICSCSS).*
- Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.
- Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.
- Diederik P Kingma. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.
- Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, et al. 2022. The stack: 3 tb of permissively licensed source code. *Preprint*.
- Achintya Kundu, Rhui Dih Lee, Laura Wynter, and Raghu Kiran Ganti. 2024. Enhancing training efficiency using packing with flash attention. *arXiv preprint arXiv:2407.09105*.
- Anton Lozhkov, Raymond Li, Loubna Ben Allal, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.
- Ziyang Luo, Can Xu, Pu Zhao, and Qingfeng Sun. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.
- Mistral AI. 2024. Announcing codestral: Unleashing the power of code models with mistral’s latest innovation. <https://mistral.ai/news/codestral/>.
- OpenAI. 2024. Gpt-4 omni: A comprehensive multimodal model for language, vision, and beyond. *arXiv preprint arXiv:2408.01234*.
- Jay M Patel and Jay M Patel. 2020. Introduction to common crawl datasets. *Getting structured data from the internet: running web crawlers/scrapers on a big data production scale*.
- Qiwei Peng, Yekun Chai, and Xuhong Li. 2024. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*.
- Nishat Raihan, Antonios Anastasopoulos, and Marcos Zampieri. 2024a. mhumaneval—a multilingual benchmark to evaluate large language models for code generation. *arXiv preprint arXiv:2410.15037*.
- Nishat Raihan, Dhiman Goswami, Sadiya Sayara Chowdhury Puspo, Christian Newman, Tharindu Ranasinghe, and Marcos Zampieri. 2024b. Cseprompts: A benchmark of introductory computer science prompts. In *International Symposium on Methodologies for Intelligent Systems*.
- Kamalkumar Rathinasamy, Ankush Kumar, Gagan Gargi, Rajab Ali Mondal, Swayam Singh, et al. 2024. Narrow transformer: Starcoder-based java-lm for desktop. *arXiv preprint arXiv:2407.03941*.
- Ricardo Rei, Nuno M Guerreiro, Daan van Stigt, Marcos Treviso, et al. 2023. Scaling up cometkiwi: Unbabel-ist 2023 submission for the quality estimation shared task. In *Proceedings of the Eighth Conference on Machine Translation*.
- Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.
- Hugo Touvron, Thibaut Lavril, et al. 2023a. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.
- Hugo Touvron, Louis Martin, et al. 2023b. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.
- Lewis Tunstall, Leandro Von Werra, and Thomas Wolf. 2022. *Natural language processing with transformers*. ” O’Reilly Media, Inc.”.
- Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.
- T Wolf. 2019. Huggingface’s transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.
- Lingling Xu, Haoran Xie, Si-Zhao Joe Qin, Xiaohui Tao, and Fu Lee Wang. 2023. Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment. *arXiv preprint arXiv:2312.12148*.
- Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, et al. 2023. Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation. *arXiv preprint arXiv:2311.08588*.
- Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*.

A Data Sources

As mentioned in Section 4.1, we carefully select a set of GitHub repositories with mojo code examples. Upon filtering, only the ones with Apache 2.0 licenses are kept. During extraction, we use BeautifulSoup¹⁰, a commonly used Python library for such purposes, to parse and extract relevant code snippets efficiently. This approach enables us to maintain consistency and accuracy in our data collection process. Table 6 includes all the repository names.

GitHub Repositories
kojunseo/mojo-wav
MoSafi2/MojoFastTrim
mattfaltyn/mojomics
igorgue/firedis
vietanhdev/chess.mojo
mzaks/mojo-prefix-sum
HJLebbink/quine-mccluskey-mojo
Deftioo/Quojo
mzaks/mojo-sort
HJLebbink/sort-networks-mojo
alainrollejr/mocodes
Irmantovani10/Stable-Diffusion.mojo
isuckatcs/advent-of-code
Sharktheone/arch-mojo
msaelices/py2mojo
PriNova/MojoPkgWorkflow
mojopaa/menv
guidorice/mojo-pytest
joelflaig/mojo-syntax
Lynet101/Mojo.community-lib
lsh/shims
mzaks/mojo-hash
rd4com/mojo-console-style-print
thatstoasty/gojo
tairov/llama2.mojo
automata/mojograd
andresnowak/Micro-Mojograd
MadAlex1997/Mojo-Arrays
endia-org/Endia
erfanzar/EasyDeL
StijnWoestenborghs/gradi-mojo
dorjeduc/momograd
basalt-org/basalt
thatstoasty/prism
thatstoasty/mog
sa-/mo-time
mojoto/morrow.mojo
Moosems/Mojo-Types
mzaks/mojo-trees
rd4com/mojo-magicalist
Benny-Nottonson/mojoDataStructures
mzaks/compact-dict
crisadamo/mojo-libc
ihnorton/mojo-ffi
thatstoasty/stump
msteele/mojo-sdl
rectalogic/mojo-qt
helehex/infrared
leandrocamos/specials
gabrieldemarmiesse/mojo-stdlib-extensions
saviorand/lightbug_http
taalhaataahir0102/Jpeg-Decoder
rd4com/mojo-learning
VMois/mojo-gym

Table 6: List of GitHub repositories for Mojo-related projects

Additional resources include webpages with documentation, tutorials, and blogs. Again, we only consider the publicly available ones for scraping. Table 7 includes the other resources used during the compilation of Mojo-Corpus.

¹⁰<https://beautiful-soup-4.readthedocs.io/en/latest/>

Webpages that feature Mojo
modular.com/mojo-programming
docs.modular.com/mojo-manual
github.com/modularml/mojo
codecademy.com/learn/mojo-programming
en.wikipedia.org/wiki/Mojo_(programming_language)
datacamp.com/community/tutorials/mojo-programming
blog.logrocket.com/getting-started-mojo-programming
modular.com/products/mojo
docs.modular.com/get-started-with-mojo
tutorialspoint.com/mojo_programming/index.htm
learnmojo.org
medium.com/tag/mojo
freecodecamp.org/news/what-is-mojo-programming-language
geeksforgeeks.org/mojo-programming-language
mojo.programming.docs.example.com
udemy.com/course/mojo-programming
coursera.org/specializations/mojo-programming
edx.org/course/introduction-to-mojo
lynda.com/Mojo-tutorials/Welcome-to-Mojo/2816042-2.html
khanacademy.org/computing/mojo-programming
academy.modular.com/mojo-programming
stackoverflow.com/questions/tagged/mojo-programming
dzone.com/articles/an-introduction-to-mojo-programming
realpython.com/mojo-pythonic-programming
pluralsight.com/courses/mojo-fundamentals
dev.to/t/mojo
hackerrank.com/domains/tutorials/mojo-programming
codewars.com/kata/search/mojo
vimeo.com/ondemand/mojoprogramming
ibm.com/cloud/learn/mojo-programming
oracle.com/mojo-programming
mojo.developer.com
guru99.com/learn-mojo-quick-guide.html
mojolanguage.school
news.ycombinator.com/item?id=31259347
linkedin.com/learning/mojo-programming-essentials
mojodojo.dev/guides/builtins/BuiltinList.html
mojohub.io/docs/tutorial
fossbytes.com/mojo-programming-tutorial
mozilla.org/mojo-learn
programiz.com/mojo-programming
packtpub.com/product/mojo-programming-cookbook
tutorialsteacher.com/mojo-programming
learnprogramming.com/mojo-language
freecodingcamp.org/mojo-programming
tutorialguru.com/intro-to-mojo-programming
coursehero.com/mojo-language-programming

Table 7: List of Mojo Programming Resources

B Filltering Steps

- F_1 Ensures licensing compliance by removing non-Apache 2.0 content, reducing the corpus to 42,245,342 tokens.
- F_2 Focuses on Mojo-specific content by excluding Python-related snippets (e.g., "python or def"), resulting in 21,973,419 tokens. This step is necessary as Python and Mojo are frequently compared online.
- F_3 Enforces structural integrity by retaining samples with at least 3 code/text blocks, each containing at least 3 characters, reducing the corpus to 15,410,741 tokens.
- F_4 Addresses repetition by filtering out samples with over 30% duplicate paragraphs or 20% duplicate characters, leaving 10,112,466 tokens.
- F_5 Removes inter-sample duplicates to ensure uniqueness and reduce bias, further reducing the corpus to 7,145,333 tokens.
- F_6 Applies language filtering using fastText (Bojanowski et al., 2017), retaining English text with a confidence threshold of 0.4, yielding the final corpus, MC*, with 6,583,948 tokens.

C Prompt Translation and Evaluation

The pseudocode version of the workflow is presented in Figure 2.

Algorithm 1 Prompt Translation and Evaluation Workflow

```

1: for each extracted prompt from the HumanEval dataset
   do
2:   for each translation system do
3:     for each target language do
4:       if the language is supported then
5:         Generate 5 translated candidate prompts
6:         Perform back translation for each candidate
7:         Compute BERT_Score and CometKiwi for
           all candidates
8:         Calculate the average score of BERT_Score
           and CometKiwi
9:         Select the best prompt based on the highest
           average score
10:      else
11:        Perform back translation
12:        Compute only the BERT_Score
13:        Select the best prompt based on BERT_Score
14:      end if
15:    end for
16:  end for
17: end for

```

C.1 Evaluation Metric 1: BERTScore

BERTScore leverages pre-trained BERT embeddings to quantify similarity between candidate and reference translations. For a candidate sentence C and a reference sentence R , let E_C and E_R denote their respective token embedding sets. The similarity score $S(i, j)$ between tokens i and j is defined by the cosine similarity of their embeddings:

$$S(i, j) = \frac{e_{C_i} \cdot e_{R_j}}{\|e_{C_i}\| \|e_{R_j}\|} \quad (10)$$

Precision P , recall R , and F1-score $F1$ are computed as follows:

$$P = \frac{1}{|E_C|} \sum_{e_{C_i} \in E_C} \max_{e_{R_j} \in E_R} S(i, j) \quad (11)$$

$$R = \frac{1}{|E_R|} \sum_{e_{R_j} \in E_R} \max_{e_{C_i} \in E_C} S(j, i) \quad (12)$$

$$F1 = 2 \cdot \frac{P \cdot R}{P + R} \quad (13)$$

Here, P represents the average maximum similarity of each token in C to any token in R , while R denotes the average maximum similarity of each token in R to any token in C . The $F1$ score is derived as the harmonic mean of precision and recall.

C.2 Evaluation Metric 2: CometKiwi

CometKiwi (Knowledge Integration via Weighted Importance) assesses translations using human judgment scores in a reference-free context, integrating linguistic features and contextual embeddings. For a source sentence $\mathbf{x}$ and candidate translation $\mathbf{y}$, CometKiwi employs a neural network $\mathcal{N}$ to map inputs to a quality score $Q(\mathbf{x}, \mathbf{y})$. The network is trained on human-annotated scores $Q_{\text{human}}(\mathbf{x}, \mathbf{y})$.

The quality score is defined as:

$$Q(\mathbf{x}, \mathbf{y}) = f(\mathbf{E}_{\text{src}}(\mathbf{x}), \mathbf{E}_{\text{cand}}(\mathbf{y}), \mathbf{L}(\mathbf{x}, \mathbf{y})) \quad (14)$$

where:

- $\mathbf{E}_{\text{src}}(\mathbf{x})$ is the source sentence embedding,
- $\mathbf{E}_{\text{cand}}(\mathbf{y})$ is the candidate translation embedding,
- $\mathbf{L}(\mathbf{x}, \mathbf{y})$ represents linguistic features.

Formally, f is expressed as:

$$f(\mathbf{E}_{\text{src}}(\mathbf{x}), \mathbf{E}_{\text{cand}}(\mathbf{y}), \mathbf{L}(\mathbf{x}, \mathbf{y})) = \mathcal{N}(\mathbf{E}_{\text{src}}, \mathbf{E}_{\text{cand}}, \mathbf{L}) \quad (15)$$

The network $\mathcal{N}$ is optimized to minimize the loss function $\mathcal{L}$:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^N (Q(\mathbf{x}_i, \mathbf{y}_i) - Q_{\text{human}}(\mathbf{x}_i, \mathbf{y}_i))^2 \quad (16)$$

where N denotes the number of training samples.

D Final Set of Hyperparameters

Both the pretraining and finetuning hyperparameters are chosen through a rigorous and iterative experimental process. We conduct a series of controlled experiments, systematically varying parameters such as learning rate, batch size, optimizer settings, and regularization factors, to identify the configuration that yields the most stable and robust performance. The final hyperparameter settings for the pretraining phase are detailed in Table 8.

Similarly, after extensive experimentation with different configurations, we arrive at the optimal settings for finetuning on the `Mojo-SFT` dataset, which are presented in Table 9. For finetuning on the `Mojo-mSFT` dataset, Table 10 outlines the hyperparameters used. This ensures that our models are finely tuned for each specific training stage, balancing performance improvements with computational efficiency.

D.1 Pretraining HyperParameters

Hyperparameter	Value
Per device train batch size	32
Gradient accumulation steps	8
Number of training epochs	3
Learning rate	5×10^{-6}
FP16	False
BF16	True
Dataloader num workers	4
Gradient checkpointing	True
Logging steps	100
DDP find unused parameters	False
Max gradient norm	1.0
Warmup steps	500
Evaluation strategy	steps
Evaluation steps	10,000
Save strategy	steps
Save steps	10,000
Save total limit	3
Load best model at end	True
Metric for best model	loss
Greater is better	False

Table 8: Final set of hyperparameters, chosen empirically after several iterations of trial and error, for pretraining on the `Mojo-Corpus`.

D.2 Finetuning Hyperparameters

Parameter	Value
Max Sequence Length	2048
Batch Size (Train/Eval)	8
Gradient Accumulation Steps	4
Number of Epochs	3
Learning Rate	$5e-5$
Weight Decay	0.02
Warmup Steps	10%
Optimizer	AdamW (8-bit)
LR Scheduler	Cosine
Precision	BF16
Evaluation Strategy	Steps
Evaluation Steps	50
Save Strategy	Steps
Save Steps	Varies
Seed	42

Table 9: Final set of hyperparameters, chosen empirically after several iterations of trial and error, for finetuning on `Mojo-SFT`.

Parameter	Value
Max Sequence Length	2048
Batch Size (Train/Eval)	16
Gradient Accumulation Steps	4
Number of Epochs	3
Learning Rate	$1e-5$
Weight Decay	0.04
Warmup Steps	15%
Optimizer	AdamW (8-bit)
LR Scheduler	Cosine
Precision	BF16
Evaluation Strategy	Steps
Evaluation Steps	100
Save Strategy	Steps
Save Steps	Varies
Seed	42

Table 10: Final set of hyperparameters, chosen empirically after several iterations of trial and error, for finetuning on `Mojo-mSFT`.

E Mojo-Coder - Examples

E.1 Sample Task 1

Difficulty: Easy

Task: Write a Mojo Code snippet that prints 'Hello World'

Verdict: PASSED

Generated Response:

```
1 fn main():
2     # Print the string "Hello, World!" to the console
3     print("Hello, World!")
4
5 # Call the main function to execute the program
6 main()
```

E.2 Sample Task 2

Difficulty: Easy

Task: Write a Mojo Code snippet that can detect Leap Year.

Verdict: PASSED

Generated Response:

```
1 fn is_leap_year(year: Int) -> Bool:
2     # Check if the year is divisible by 4
3     if year % 4 == 0:
4         # If it's divisible by 4, check if it's not divisible by 100
5         if year % 100 != 0:
6             return True
7         # If it's divisible by 100, check if it's divisible by 400
8         elif year % 400 == 0:
9             return True
10        # If it's divisible by 100 but not by 400, it's not a leap year
11        else:
12            return False
13    # If the year is not divisible by 4, it's not a leap year
14    else:
15        return False
16
17 # Example usage
18 fn main():
19     let test_year = 2024
20     let result = is_leap_year(test_year)
21
22     if result:
23         print(test_year, "is a leap year")
24     else:
25         print(test_year, "is not a leap year")
```

E.3 Sample Task 3

Difficulty: Medium

Task: Write a Mojo Code snippet to do Radix Sort.

Verdict: PASSED

Generated Response:

```
1 from math import max
2
3 fn radix_sort(inout arr: DynamicVector[Int]):
4     """
5     Sort the given array using radix sort algorithm.
6     """
7     if len(arr) <= 1:
8         return
9
10    # Find the maximum element to determine the number of digits
11    max_element = arr[0]
12    for i in range(1, len(arr)):
13        if arr[i] > max_element:
14            max_element = arr[i]
15
16    # Determine the number of digits
17    max_digits = 0
18    while max_element > 0:
19        max_digits += 1
20        max_element //= 10
21
22    # Perform counting sort for every digit
23    for digit_place in range(max_digits):
24        counting_sort(arr, digit_place)
25
26 fn counting_sort(inout arr: DynamicVector[Int], digit_place: Int):
27     """
28     Perform counting sort for a specific digit position.
29     """
30     n = len(arr)
31     output = DynamicVector[Int](n)
32     count = DynamicVector[Int](10, 0)
33
34     # Count occurrences of each digit
35     for i in range(n):
36         index = (arr[i] // (10 ** digit_place)) % 10
37         count[index] += 1
38
39     # Change count[i] so that count[i] now contains actual
40     # position of this digit in output[]
41     for i in range(1, 10):
42         count[i] += count[i - 1]
43
44     # Build the output array
45     i = n - 1
46     while i >= 0:
47         index = (arr[i] // (10 ** digit_place)) % 10
48         output[count[index] - 1] = arr[i]
49         count[index] -= 1
50         i -= 1
51
52     # Copy the output array to arr[], so that arr[] now contains sorted numbers
53     for i in range(n):
54         arr[i] = output[i]
55
56 # Example usage
57 fn main():
58     var arr = DynamicVector[Int]()
59     arr.push_back(170)
60     arr.push_back(45)
61     arr.push_back(75)
62     arr.push_back(90)
63     arr.push_back(802)
64     arr.push_back(24)
65     arr.push_back(2)
66     arr.push_back(66)
67     arr.push_back(110)
68     arr.push_back(1)
69
70     print("Original array:", arr)
71     radix_sort(arr)
72     print("Sorted array:", arr)
```

E.4 Sample Task 4

Difficulty: Hard

Task: Write a function in Mojo that takes a 2D grid where 1 represents land and 0 represents water. An island is a group of 1s connected horizontally or vertically. The function should return the maximum area of an island in the grid. If there are no islands, return 0.

Verdict: **FAILED** - Does not pass all the test cases.

Generated Response:

```
1 from python import Python
2
3 fn max_area_of_island(grid: PythonObject) -> Int:
4     """
5     Find the maximum area of an island in a 2D grid.
6
7     Args:
8     grid: A 2D list of integers representing the grid, where 0 is water and 1 is land.
9
10    Returns:
11    The maximum area of an island.
12    """
13
14    # Get the dimensions of the grid
15    rows = len(grid)
16    if rows == 0:
17        return 0
18    cols = len(grid[0])
19
20    # Initialize max_area to 0
21    max_area = 0
22
23    # Define a function to explore an island and calculate its area
24    fn explore_island(i: Int, j: Int) -> Int:
25        if i < 0 or i >= rows or j < 0 or j >= cols or grid[i][j] == 0:
26            return 0
27        # Mark the current cell as visited (set to 0)
28        let current_area = 1
29        grid[i][j] = 0
30        # Explore adjacent cells
31        current_area += explore_island(i + 1, j)
32        current_area += explore_island(i - 1, j)
33        current_area += explore_island(i, j + 1)
34        current_area += explore_island(i, j - 1)
35        return current_area
36
37    # Iterate through each cell in the grid
38    for i in range(rows):
39        for j in range(cols):
40            # If a land cell is found, explore its area
41            if grid[i][j] == 1:
42                let current_area = explore_island(i, j)
43                max_area = max(max_area, current_area)
44
45    return max_area
46
47 fn main():
48     # Example usage
49     let grid = Python.list([
50         Python.list([1, 0, 0, 0, 0]),
51         Python.list([0, 1, 0, 0, 1]),
52         Python.list([0, 0, 1, 0, 1]),
53         Python.list([0, 0, 0, 1, 0]),
54         Python.list([1, 0, 0, 0, 1])
55     ])
56
57     let result = max_area_of_island(grid)
58     print("Maximum area of an island:", result)
59
60 main()
```

F Base Models vs Mojo

As mentioned in Section 4.4.1, off-the-shelf models often struggle to generate working code snippets in Mojo, when prompted. Even SOTA models like GPT-4o and Claude severely underperform when the prompt is non-English. We present some examples here reporting such issues.

F.1 Sample Task 1

Difficulty: Easy

Task: Write a Mojo Code snippet that prints 'Hello World'

GPT-4o

Verdict: **FAILED** - Parsing Error

Generated Response:

```
fn main() {  
    print("Hello World")  
}  
main()
```

Claude-3.5-Sonnet

Verdict: **PASSED**

Generated Response:

```
fn main():  
    print("Hello World")
```

WizardCoder

Verdict: **FAILED** - Syntax Error

Generated Response:

```
import logging  
  
logging.info("Hello, World!")
```

CodeGemma

Verdict: **FAILED** - Compilation Error

Generated Response:

```
@Mojo( name = "hello-world")  
public class HelloWorldMojo  
extends AbstractMojo {  
  
    public void execute() throws  
        MojoExecutionException {  
        getLog().info("Hello World");  
    }  
  
}
```

CodeLLaMA

Verdict: **FAILED** - Compilation Error

Generated Response:

```
<plugin>
  <groupId>com.example</groupId>
  <artifactId>helloworld
</artifactId>
  <version>1.0.0</version>
  <executions>
    <execution>
      <phase>validate</phase>
      <goals>
        <goal>hello-world
        </goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Mistral

Verdict: **FAILED** - Compilation Error

Generated Response:

```
#include <stdio.h>

int main() {
    printf("Hello World");
    return 0;
}
```

F.2 Sample Task 2

Difficulty: Easy

Task: Write a Mojo Code snippet that can detect Leap Year.

GPT-4o

Verdict: PASSED

Generated Response:

```
fn is_leap_year(year: Int) -> Bool:
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        return true
    else:
        return false

fn main():
    let year = 2024 # Change this value to test other years
    if is_leap_year(year):
        print(f"{year} is a leap year")
    else:
        print(f"{year} is not a leap year")

main()
```

Claude-3.5-Sonnet

Verdict: PASSED

Generated Response:

```
fn is_leap_year(year: Int) -> Bool:
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)

fn main():
    let current_year = 2024
    print("Is", current_year, "a leap year?", is_leap_year(current_year))

    # Test a few more years
    for year in range(2020, 2026):
        print(year, "is", "a leap year" if is_leap_year(year) else "not a leap year")

main()
```

WizardCoder

Verdict: FAILED - Compilation Error

Generated Response:

```
var leapYearApplet = {
    activate: function(args) {
        var year = args.year;

        // Check if the year is a leap year
        if ((year % 4 === 0) && (year % 100 !== 0) || (year % 400 === 0)) {
            Mojo.log("The year " + year + " is a leap year.");
        } else {
            Mojo.log("The year " + year + " is not a leap year.");
        }
    }
};
```

CodeGemma

Verdict: **FAILED** - Compilation Error

Generated Response:

```
// Mojo code to detect leap year
int year = 2024;

// Check if the year is divisible by 4
if (year % 4 == 0) {
    // Check if the year is divisible by 100
    if (year % 100 == 0) {
        // Check if the year is divisible by 400
        if (year % 400 == 0) {
            // The year is a leap year
            print("The year " + year + " is a leap year");
        } else {
            // The year is not a leap year
            print("The year " + year + " is not a leap year");
        }
    } else {
        // The year is a leap year
        print("The year " + year + " is a leap year");
    }
} else {
    // The year is not a leap year
    print("The year " + year + " is not a leap year");
}
```

CodeLLaMA

Verdict: **FAILED** - Compilation Error

Generated Response:

```
$year = 2020;
if (date('L', mktime(0, 0, 0, 1, 1, $year)) == 1) {
    echo "It's a leap year!";
} else {
    echo "It's not a leap year.";
}
```

Mistral

Verdict: **FAILED** - Compilation Error

Generated Response:

```
#!/usr/bin/perl
use strict;
use warnings;

# Function to check if a number is prime
sub is_prime {
    my ($num) = @_;
    if ($num <= 1) {
        return 0; # 0 and 1 are not prime
    }
    for (my $i = 2; $i * $i <= $num; $i++) {
        if ($num % $i == 0) {
            return 0; # if divisible, not prime
        }
    }
    return 1; # if no divisors, prime
}
```