Direct Preference Optimisation with Compiler Feedback: Your Language Model Is Secretly a Compiler

Authors: Harshal Jain, Phillippe Guyard, Vishrut Malik, Marat Davudov, Giles Billenness

Abstract

Programming serves as a potent and ubiquitous problem-solving tool. Enhancing its productivity and accessibility through systems capable of aiding programmers or even autonomously generating code stands as a significant goal. However, integrating AI innovations into this domain has presented challenges. While recent advancements in large language models (LLMs) showcase remarkable code generation capabilities for simpler tasks, they often falter when tasked with producing correct code in compiled languages like C++. In this paper, we propose a straightforward approach that leverages synthetic preference data obtained via compiler feedback to train existing language models in writing compilable code. Our method uses Direct Preference Optimisation (DPO) to improve the performance of a wide range of models. It also consistently surpasses traditional supervised fine-tuning techniques (SFT). Altogether, in terms of compilation success, we achieve improvements ranging 48% to 234% compared to base models, and 6% to 154% compared to SFT. The source code for our project can be found at https://github.com/Philippe-Guyard/COMP0087

1 Introduction

The quality and efficiency of code generation within software development play increasingly pivotal roles in improving the performance of software applications (Smith and Johnson, 2018). Leveraging machine learning techniques, particularly reinforcement learning (RL), to fine-tune Large Language Models (LLMs) via preference feedback has been a developing field (Brown et al., 2020). A common approach considers human-inputted feedback to generate preference data for different code generations; however, using compiler feedback instead (Zhang et al., 2019) presents an opportunity to ultimately advance the state-of-the-art in automated code generation techniques, facilitating the development of more reliable and efficient software systems.

In this paper, we propose a novel approach that leverages synthetic preference data obtained via Direct Preference Optimisation (DPO) on compiler feedback to enhance the performance of popular LLMs in the domain of C++ code generation. We show that by leveraging compiler feedback, we can obtain traditionally expensive preference data for free and use it for preference optimisation algorithms. Our method also has the advantage of being entirely objective, an attribute typically lacking in human preference. Our DPO-based approach ensures that generated code meets the fundamental requirement of compilation, thus aligning more closely with real-world software development needs. We explore the use of compiler feedback as preference data across a range of LLMs, including general models, instructional models, and code-specific models. We also compare our DPO-based method to traditional fine-tuning techniques, specifically Supervised Fine-Tuning (SFT) and observe its effects on the performance of each LLM.

2 Related Work

Recently, research has focused on pre-training language models on the extensive code corpus from open-source repositories (Zan et al., 2023; Niu et al., 2022) to improve code generation tasks. These include CodeGPT (Lu et al., 2021), a comparable decoder-only GPT model introduced alongside the CodeXGLUE benchmark and CodeBERT (Feng et al., 2020), which conducts encoder-only pre-training using Masked Language

Modeling (MLM) and Replaced Token Detection tasks. However, while relying heavily on self-supervised objectives for text generation, these pre-trained models struggle to maintain essential sequence-level code attributes, such as syntactic and operational accuracy in the generations.

Conversely, much effort has been made to advance the development of code generation (Li et al., 2018) by leveraging structure. This includes using different logical forms of code, such as the abstract syntax tree (AST) (Kim et al., 2021), sketch (Nye et al., 2019) and graph (Yasunaga and Liang, 2021). However, despite these efforts, many of these structure-aware code generation models still struggle to ensure the syntactic and operational accuracy of the generated codes. These models are not optimised for non-differentiable code-specific objectives such as compilability, readability, or passing test cases. Consequently, this results in performance deficiencies.

In contrast to tokens in natural language, generated code must follow certain specific code-related characteristics. This covers both syntactic and functional correctness since machine execution of the resulting code depends on passing compilation and unit tests. RL-based fine-tuning mechanisms (Wang et al., 2022; Le et al., 2022) are used to enhance the quality of generated code by Programming Language (PL) models. For example, Le et al. (2022) has recently studied the integration of RL with unit test signals in the fine-tuning of the program synthesis models. PPOCoder (Shojaee et al., 2023) utilises non-differentiable feedback from code execution and structure alignment to integrate external code-specific knowledge, i.e. compiler feedback with code structure elements, into the model optimisation process. Thus, one of the limitations of Shojaee et al. (2023) is the added computational time required for RL-based optimisation and the possibility that it may not yield significant improvements on other evaluation metrics not directly targeted during RL optimisation. In our case, to counter this added computational inefficiency, we incorporate compiler feedback into our optimisation process using DPO, which is a different approach to training language models from human preferences without using reinforcement learning (RL).

On the other hand, advanced PL models like Alphacode (Li et al., 2022), claimed to have outperformed half of the human competitors in real-world programming competitions, incorporate a multi-step fine-tuning process: a separate fine-tuning dataset, tempering, GOLD (Pang and He, 2021), value conditioning and prediction. But these improved models focus on solving much harder and more complex problems, whereas we aim for a simpler objective of improving models by training them how to write compilable code.

3 Methodology

This section will outline the theoretical background of the methods employed to align our chosen LLMs to the desired output on code completion tasks and test our hypothesis.

3.1 Dataset

The dataset used for evaluation is the Project CodeNet C++ 1000 Benchmark (Puri et al., 2021). This dataset consists of C++ code submissions for problems from various competitions, totalling 1000 problems. It was also one of the source datasets of the AlphaCode paper (Holtzman et al., 2020).

In the Project CodeNet dataset, submissions can have different statuses depending on whether they correctly solved the given problem, exceeded the time limit, compiled, etc. Since for our purposes, we do not require such fine-grained filtering and are only interested in correct C++ code, we are filtering by the "Accepted" status and using subsets of these submissions for fine-tuning and evaluation.

3.2 Masking

For each C++ program with n symbols, we remove a certain fraction m of the symbols from the end of the program, where $0 \leq m \leq 1$. This masking strategy aims to simulate a typical completion scenario, where a developer submits an incomplete program to the LLM and expects a finished result.

3.3 Supervised Fine Tuning

The initial phase of the model alignment procedure involves acquiring a dataset consisting of prompt-response pairs (Ouyang et al., 2022). These pairs serve as direct input for fine-tuning the model, employing the same next-token prediction training objective utilised during pre-training

to establish the foundation of the model (Wolfe, 2023). To enhance resource efficiency in this step, we adopt the parameter-efficient fine-tuning method known as LoRA (Hu et al., 2021). Specifically, as we utilise QLoRA (Dettmers et al., 2023) an extension of the LoRA algorithm.

Instead of optimising the full set of model parameters Φ using the original conditional language modelling objective (equation 1), LoRA aims to only optimise the much smaller set of task-specific parameters Θ, using the objective described by (equation 2). This significantly reduces the computational costs as the gradient dimensionality is reduced from $|\Delta\Phi| = |\Phi|$ to $|\Delta\Phi(\Theta)| = |\Theta|$, and $|\Theta| \ll |\Phi|$ (Hu et al., 2021).

Conditional language modelling objective (Hu et al., 2021):

$$\max_{\Phi} \sum_{(x,y)\in Z} \sum_{t=1}^{|y|} \log\left(P_{\Phi}(y_t|x, y_{<t})\right) \quad (1)$$

and LoRA objective (Hu et al., 2021):

$$\max_{\Theta} \sum_{(x,y)\in Z} \sum_{t=1}^{|y|} \log\left(p_{\Phi_0+\Delta\Phi(\Theta)}(y_t|x, y_{<t})\right) \quad (2)$$

At this moment, even "small" LLMs have several billions of parameters. Due to this, even with a significantly reduced number of trainable parameters with LoRA, the fine-tuning process can consume more memory than a single GPU has available. QLoRA is a technique that addresses this issue by further increasing memory efficiency without sacrificing performance (Dettmers et al., 2023). QLoRA reduces the memory required for training by applying quantisation to LoRA matrices. Quantisation discretises the weights from a representation with high information entropy to a representation with low information entropy. The 4-bit NormalFloat Quantisation used by QLoRA not only produces quantised weights that take up less memory but also allows de-quantisation after training to restore r-the original precision (Dettmers et al., 2023).

3.4 Direct Preference Optimisation

The second and final step of LLM alignment is Reinforcement Learning from Human Feedback (RLHF) (Ouyang et al., 2022). This step is split into two phases: the Reward Modelling phase and the RL Fine-Tuning phase (Rafailov et al., 2023).

The Reward Modelling phase aims to estimate a latent reward function $r^*(y|x)$ that generates human labellers' preference given a pair of model outputs (y_1, y_2) (Rafailov et al., 2023).

The learned reward function $r_\phi(y|x)$ is then used in the RL Fine-Tuning phase to provide feedback to the model. Due to the discrete nature of language modelling, the resulting optimisation objective is not differentiable and is optimised with Reinforcement Learning by constructing the reward function $r(y|x) = (r_\phi(y|x) - \beta \log \pi_\theta(y|x) - \log \pi_{\text{ref}}(y|x)$ and optimised using Proximal Policy Optimisation (Schulman et al., 2017). Base preference policy π_{ref} and language model policy π_θ is initialised to the starting SFT model π^{SFT}, with β controlling deviation from π_{ref} (Rafailov et al., 2023).

Although the RLHF approach effectively produces well-aligned models, it is considerably more complex than supervised fine-tuning and is associated with significant computational costs. To overcome these issues Rafailov et al. (2023) proposed an alternative RL-free method, the Direct Prefence Optimisation (DPO) algorithm, which implicitly optimises the same objective as RLHF using a simple cross-entropy loss objective.

Specifically, DPO minimises the following objective, where the expectation is under $(x, y_w, y_l) \sim \mathcal{D}$, with (y_w, y_l) denoting the completion pair of 'winning' and 'losing' preferences:

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}\left[\log \sigma\left(\beta \log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right] \quad (3)$$

Rafailov et al. (2023) outlines the DPO algorithm as follows: (1) Construct a preference dataset $\mathcal{D} = \{x^{(i)}, y_w^{(i)}, y_l)^{(i)}\}_{i=1}^N$ by assigning labels to models responses based on human preference, such that $y_1, y_2 \sim \pi_{\text{ref}}(\cdot \mid x)$ for each prompt x. (2) Optimise the language model policy π_θ to minimise $\mathcal{L}_{\text{DPO}}$ for given π_{ref} and desired β.

This algorithm allows for training on preference data directly and using simple cross-entropy loss, providing RL-free model alignment and avoiding associated heavy computational costs. The authors of the paper (Rafailov et al., 2023) have also demonstrated that DPO is at least as effective as

existing RLHF approaches, including PPO-based methods, which are used in the original LLM alignment framework (Ouyang et al., 2022).

4 Models

Several Large language models (LLMs) have recently been released in open-weights/code form, offering state-of-the-art performance and downstream fine-tuning on many text-based tasks, such as code completion.

4.1 LLMs

The LLMs selected in this paper are diverse, ranging from base to instruct and code-specific models, increasing the comprehensiveness and value of the experiments conducted. They also demonstrate promising performance for code generation tasks, which should directly impact the performance we see from our experiments. The models were also available in quantised form provided by unsloth (Han and Han, 2024), allowing for a more flexible experiment environment.

4.2 Code Llama-7b

This text-generation model, released in August 2023 (Rozière et al., 2024), is fine-tuned from the Llama 2 7B parameter model (Touvron et al., 2023), released in July 2023. Compared to GPT-3.5 (OpenAI et al., 2024), PaLM (Chowdhery et al., 2022), and StarCoder (Li et al., 2023), this model presented promising performance in HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) code synthesis tasks. This model was not trained to require a chat template or work well with long-form natural language instructions.

4.3 Mistral-7b-instruct

This chat-based model is fine-tuned from the Mistral 7B parameter model (Jiang et al., 2023), released at the same time in September 2023. Utilising the same memory efficiency techniques that resulted in Mistral-7b achieving similar results in the HumanEval (0-shot) and MBPP (3-shot) code tasks to Code-Llama 7B. This instruct model is trained to require prompt templates that define the instruction, relevant data and the LLM's role. By default, the model responds to prompts in a chat-based form that would require model-specific parsing to retrieve relevant code. Performance was improved by using 1 shot prompts with an example of the desired output.

4.4 Gemma-7b

This text-generation model, released in February 2024 (Team et al., 2024), boasted improved performance on the HumanEval and MBPP Python code synthesis tasks compared to Mistral 7B, Llama 2 13B, and CodeLLaMA-7B. Similar to Code Llama-7b, this model is not instruct fine-tuned.

4.5 unsloth

The unsloth project provides faster and more memory-efficient QLoRA and LoRA fine-tuning of popular LLMs. This is primarily through the use of mixed precision computation with manually optimised gradient computation, Triton kernels (Tillet et al., 2019), and Flash Attention-2 (Dao, 2023). unsloth provides pre-quantised versions of many popular LLMs hosted on huggingface (Wolf et al., 2020) that were used in this paper.

4.6 Prompts

The two base models (Gemma and CodeLlama) that we used did not require any prompting and worked by just sending the raw uncompleted code to them, for example:

```
USER:
\#include <iostream>

int main() {
    std::cout << "Hello,
MODEL:
    World!" << std::endl;
    return 0;
}
```

In this case, parsing the output code was very easy, as it was just the entirety of the text outputted by the model

For the mistral instruct model, this approach did not work, as the model would often output extra text, which would harm compilation, for example:

```
USER:
\#include <iostream>

int main() {
    std::cout << "Hello,
MODEL:
    World!" << std::endl;
    return 0;
}
Your code had an unfinished line. Here is the
     completion.
```

Two techniques were used to address this:

1. Adding a system prompt that precisely defines the model response format

2. Adding a ‘Hello, World!’ code completion example, following a 1-shot prompting style for Mistral-7B-it.

This produced the following prompt format:

```
<s>[INST] You are an assistant that helps users
     with writing compiler-friendly C++ programmes.
     Your outputs should be exclusively C++
     programmes that can be compiled with C++17.
     Please make sure to delimit your code with
     ```. Here is an example:
```cpp
#include <iostream>

using namespace std;

int main() {{
    cout << "Hello,
```
 [/INST]
```cpp
#include <iostream>

using namespace std;

int main() {{
    cout << "Hello, World! << endl;
}}
```
</s>[INST] ```cpp
#include <bits/stdc++.h>

#define REP(i, n) for(int i=0;i<(int)(n);i++)
#define ALL(x) (x).begin(),(x).end()
const int INF = 1e9;

using namespace std;
int main(){
    REP(i, 9) REP(j, 9)
        cout << i+1 << "x" << j+1 << "=" << (i+1) *
             (j+
```
[/INST]
```

Note that the <s>, </s>, [INST], [/INST] tokens here come from using HuggingFace chat templates (HuggingFace, 2024).

In this case, the model output was:

```
```cpp
#include <bits/stdc++.h>

#define REP(i, n) for(int i=0;i<(int)(n);i++)
#define ALL(x) (x).begin(),(x).end()
const int INF = 1e9;

using namespace std;
int main(){
    REP(i, 9) REP(j, 9)
        cout << i+1 << "x" << j+1 << "=" << (i+1) *
             (j+1) << endl;
    return 0;
}
```
```

To parse the output, we find the 8^{th} and 9^{th} occurrences of ` ` `, and extract everything between ` ` `cpp and ` ` `. Empirically, we found that using delimiters of the form ` ` `cpp and ` ` ` worked best. We also attempted to start and end with #####, but the base model kept outputting delimiters like ` ` `cpp and ` ` `.

5 Experiments

This section describes the experimental pipeline we designed to evaluate the impact of SFT and DPO on the ability of LLMs to complete C++ code.

5.1 Computational Resources

All experiments were run on the UCL Department of Computer Science lab machines equipped with NVIDIA GeForce RTX 3090 Ti GPUs with 24GB VRAM (TSG, 2023). Attempts were made to run the experiments on the UCL Myriad HPC (UCL, 2024), which provides more powerful GPUs and would allow to train larger models, however, the long job queue times made this approach infeasible under the time constraints of this project.

5.2 Dataset

To form the training and evaluation datasets, we are using submissions from all 1000 problems available from the Project CodeNet C++ 1000 benchmark. For each problem, we filter out C++ submissions with "Accepted" status. From this subset, we select 5 distinct submissions for the training dataset and 2 distinct submissions for the evaluation dataset, totalling 5000 samples for the training set and 2000 samples for the testing set. At this stage, we also ensure that all the samples we use compile without errors on our system.

We evaluate all models on the test set at each stage of fine-tuning: (1) Base, (2) SFT, and (3) DPO. The training set is used for SFT on all models. The procedure for generating preference datasets for DPO is described in Section 5.4.

5.3 SFT Step

We initialise the models with unsloth and train with the SFTTrainer class from Hugging Face using the training dataset. We start with a learning rate of $2e^{-4}$ and use a linear schedule to decay the value to approximately $2e^{-7}$. The optimiser is 8bit AdamW. Finally, we employ a weight decay of 0.01.

As for QLoRA parameters, Dettmers et al. (2023) find that the lora_r parameter is unrelated to the final performance of the model if LoRA is used on all layers, which is the case for our implementation. Based on this, we use a value of lora_r to 64, as in original the QLoRA paper (Dettmers et al., 2023). For gemma7B, we ran

into memory issues when trying to run DPO with `lora_r` = 64, so we used a much lower `lora_r` = 4, for this model only. The lora_alpha parameter for all models was chosen to stay in line with the standard LoRA scaling factor $\frac{\alpha}{r} = 1$.

5.4 DPO Step

After the SFT step has been performed on a model, we perform DPO. To form the DPO preference dataset, we evaluate the SFT model on the training dataset and extract all the responses that earned a "Bad" score. Then, for each "Bad" response, we form an entry $(x^{(i)}, y_w^{(i)}, y_l^{(i)})$ in the preference dataset $\mathcal{D}$ by setting $x^{(i)}$ to the prompt, $y_l^{(i)}$ to the "Bad" response, and $y_w^{(i)}$ to the original unmasked submission from the Project CodeNet dataset (Figure 1).

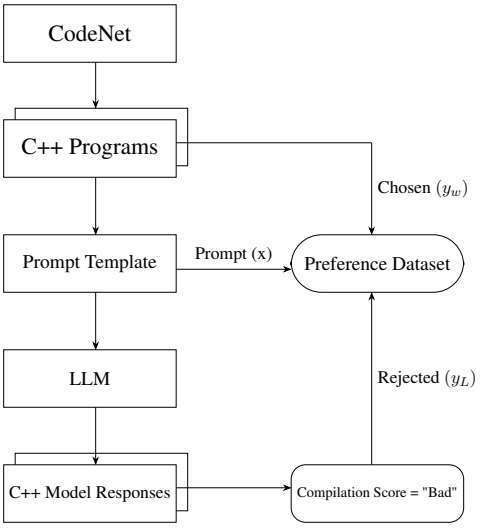

Figure 1: DPO Preference Dataset Generation

A model is then trained using this dataset using the DPOTrainer class from Hugging Face. Similarly to the SFT step, we choose a linear learning rate schedule, with a starting value $5e^{-6}$ to decay the value to approximately $6e^{-9}$. Again, we use the AdamW 8bit optimiser and a weight decay of 0.01.

For DPO-specific parameters, we use default hugging face parameters, specifically: $\beta = 0.1$, `label_smoothing` = 0 and sigmoid loss. For the reference model, we use the same architecture as the initial language model ending with a fully-connected layer (again, this is a standard hugging face procedure when no reference model is specified).

5.5 Evaluation Metric

After the completion of each training step, the resulting models are evaluated on the test set. The evaluation pipeline can be seen in Figure 2. We use greedy decoding strategy (Gu et al., 2017) for generating model responses, as other strategies, such as beam search (Graves, 2012), were prohibitively slow given our computational resources and the project's time frame.

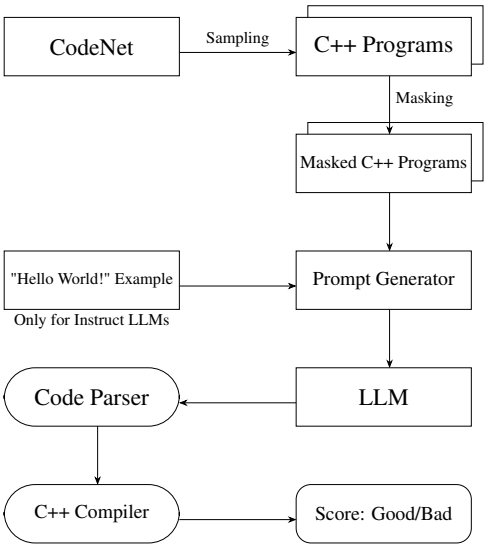

Figure 2: Evaluation Pipeline

C++ code submissions from the test set are passed through the masking procedure with $m = 0.1$, appended to a prompt and fed into the model. Depending on the type of model being tested, the prompt format is different and instruct-tuned LLMs include a "Hello, World!" completion example to specify the response format.

To assign scores to the model output, we parse the code and attempt to compile it with g++. If the compilation fails with an error, the response is given a score of 0 or "Bad". If the code successfully compiles, the response is given a score of 1 or "Good". Compilation with warnings is regarded as an unsuccessful compilation.

6 Results and Discussion

Table 1 summarises the compilation success rates across the chosen 3 LLMs and the additions of SFT and DPO steps, where 'success' is defined as a compilation with no errors.

	Gemma-7b	Mistral-7b-It	CodeLlama
Base	48.34%	27.22%	53.64%
+SFT only	61.52%	35.76%	74.70%
+DPO only	56.09%	75.83%	**79.34%**
+SFT+DPO	**72.19%**	**90.93%**	21.66%

Table 1: Performance of different models and algorithms showing compilation success rates. DPO only and SFT + DPO are the approaches presented in this paper. The best performances per LLM are bolded.

In Table 1, we observe consistent improvements from each base LLM when adding the SFT step, with CodeLlama gaining the proportionally largest increase in performance. Similarly, the DPO step improves all base LLMs, although Mistral-7b-Instruct gains a significantly larger increase in performance compared to the other LLMs, almost tripling the number of successful compilations from 27.22% to 75.83%.

We identify that the highest compilation success rates for each LLM are achieved by varied approaches of adding SFT and DPO. Gemma-7b and Mistral-7b-It both benefit the most through implementing SFT and DPO together, however CodeLlama exhibits a significant decrease in performance here and instead performs its best when only adding the DPO step; this is further explored in section 6.1.

Table 2 below highlights the relative performance of this paper's identified optimal algorithm additions across each LLM against both the base approach and the SFT-only approach.

Model	Vs. Base	Vs. +SFT
Gemma-7b (+SFT+DPO)	+49.34%	+17.34%
Mistral-7b-It (+SFT+DPO)	**+234.06%**	+154.28%
CodeLlama (+DPO)	+47.91%	**+6.21%**

Table 2: Performance improvement from base and base+SFT to the best performance for each model.

6.1 CodeLlama SFT + DPO Analysis

Remarkably, we identify a drastic drop in performance for CodeLlama below the base LLM's compilation success rate. Following manual inspection, we noticed that the preference dataset generated by CodeLlama's SFT-only model has chosen and rejected prompts that are very close to each other at the character-level, where typically the model misspells one variable name or single word.

To confirm this hypothesis, we wrote a simple Python script using the `nltk` library to measure the Levenshtein edit distance between chosen and rejected prompts in the CodeLlama preferences dataset. We noted that the average edit distance of two CodeLlama prompts is approximately twice as small as that of Gemma, as per the following results:

- CodeLlama - Average edit distance: 107.95
- Gemma (for reference) - Average edit distance: 223.82

We believe this is the same effect observed in (Pal et al., 2024). Executing DPO on a preference dataset where good and bad examples are very similar can cause, in some cases, degradation of the log probabilities of later tokens and poor performance due to the nature of DPO's loss incentive, demonstrating a flaw in the algorithm. The main difference is that (Pal et al., 2024) was trained on a corpus of mathematical problems and human feedback, as opposed to our approach of coding problems with compiler feedback, however we theorise a similar explanation since the measure of string proximity should not concern the semantics of the given problems.

Since SFT already optimises CodeLlama well (as we achieve a 74.70% compilation success rate), hence suggesting a generated preference dataset which suffers from low edit distances, we observe a performance decrease from the highest compilation rate after the SFT step, from 74.70% to 21.66%.

6.2 Mistral Performance Analysis

We note that the Mistral-7B-it model disproportionately benefitted from our fine-tuning process. We believe that there are two effects here that are superimposed to yield this boost in accuracy. As mentioned in Section 4.6, we had to write a system prompt for Mistral Instruct, as well as a parser to extract the code from the output. Our parsing of the output was not perfect. Indeed, sometimes the non-fine-tuned Mistral model would make small mistakes, such as forgetting to delimit its code with special delimiter tags, which would result in our system flagging the potentially good and compilable sample as 'Bad'. We believe that the fine-

tuning acted on Mistral's ability to obey the formatting guidelines as well as consistently generate better code. These two effects compounded are what yielded such a performance boost.

6.3 Evaluation Limitations

Stepping back, it's crucial to recognise the limitations of relying solely on compilation success rate as a measure of code generation quality. This scoring system will prefer poor-quality code, which compiles, over high-quality code, which does not compile due to a small bug. Other metrics for code quality, such as scalability, efficiency, and correctness given unit tests, are not accounted for. Nonetheless, in the context of this project, we maintain compilation success rate as an absolute upper bound for measuring code quality since high-quality code inherently assumes compilability.

7 Conclusion

In this report, we have presented two simple approaches (based on one common idea) to train Language Models how to generate compilable code for the C++ language. We improve the capabilities of a wide range of LLMs in generating valid C++ code, with model performances on the given task going up by anywhere from 48% to 234% as compared to base models, and 6% to 154% as compared with SFT. To achieve this, we leveraged existing compiler tooling for C++17 and used compiler feedback to create a synthetic preference dataset for each model. We then further trained our models with the DPO algorithm on this preference dataset (preceded by an optional SFT step). While working with DPO, we also rediscovered a flaw in the algorithm presented two months ago in (Pal et al., 2024), the difference being that our setting is code generation, whereas the original authors had this issue with mathematical problem-solving.

While we understand that focusing on code compilation rather than code correctness may be limited in its direct applications, we believe it is an important first step to better code generation of compiled languages in widely used LLMs since successful compilation is required for any code to run. The downstream applications of this research remain at the forefront of current code generation research.

In conclusion, we have achieved our aim of investigating how compiler feedback can be successfully integrated into the LLM fine-tuning process for code generation. We demonstrate that it serves well as a substitute for human preference data, significantly improving code generation for compiled languages across different types of popular LLMs.

8 Future work

There are multiple directions one could take to build on this work.

For the sake of time, we only concentrated on quantised 7B models available from (Han and Han, 2024); however, it would be interesting to see the difference in results when fine-tuning as we have on different size models and with non-quantised versions.

As discussed in Section 6.1, our process can break on some models due to a fundamental flaw of DPO. However, a similar phenomenon has already been observed by (Pal et al., 2024), suggesting an improved algorithm DPO-Positive (DPOP). It would be interesting to see whether the suggested algorithm solves the flaw we encountered of similar completion pairs.

Experiments could be conducted on the effect of different decoding methods on preference set generation for DPO and the results from our target metric. Some alternative decoding methods could include Nucleus Sampling (Holtzman et al., 2020), Contrastive search (Su et al., 2022), and Beam-search decoding (Graves, 2012). Using decoding methods that incorporate an element of randomness might avoid the issue we encountered using DPO with CodeLlama, as it would be more likely for "bad" samples to vary by a greater degree from the ground truth.

Our usage of available compiler feedback could also be extended. In this research, we only use the compiler feedback to confirm compilability. However, we could use the compiler's warnings to generate more specific preference data, perhaps fine-tuning in multiple waves e.g. first train the model to not generate errors, then train not to generate warnings, etc. Some papers have exploited more compiler feedback (Shojaee et al., 2023), which could be a source of inspiration for further work.

Finally, it would be possible to use the compiler's AST representations to synthetically generate bad samples. This would be much more involved in terms of engineering and model analysis. A simple example of such a procedure would be: (1) Notice that CodeLlama tends to make typos in variable names (2) Synthetically take good code samples and rename *some* occurrences of a variable to a similar word but misspelt (3) Use these samples as bad samples in preference data.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models.

Tom Brown et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana

Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. Palm: Scaling language modeling with pathways.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms.

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.

Alex Graves. 2012. Sequence transduction with recurrent neural networks.

Jiatao Gu, Kyunghyun Cho, and Victor O. K. Li. 2017. Trainable greedy decoding for neural machine translation.

Michael Han and Daniel Han. 2024. unsloth. `https://github.com/unslothai/unsloth`.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models.

HuggingFace. 2024. Template for chat models.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning.

Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code completion with neural attention and pointer networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, IJCAI-2018. International Joint Conferences on Artificial Intelligence Organization.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you!

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy,

Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation.

Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep learning meets software engineering: A survey on pre-trained models of source code.

Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to infer program sketches.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Kata-

rina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. Gpt-4 technical report.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback.

Arka Pal, Deep Karkhanis, Samuel Dooley, Manley Roberts, Siddartha Naidu, and Colin White. 2024. Smaug: Fixing failure modes of preference optimisation with dpo-positive.

Richard Yuanzhe Pang and He He. 2021. Text generation by learning from demonstrations.

Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. Execution-based code generation using deep reinforcement learning.

John Smith and Robert Johnson. 2018. Improving software performance through enhanced code generation techniques. *IEEE Transactions on Software Engineering*, 44(3):201–215.

Yixuan Su, Tian Lan, Yan Wang, Dani Yogatama, Lingpeng Kong, and Nigel Collier. 2022. A contrastive framework for neural text generation.

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Amélie Héliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Clément Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Christian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, Justin Mao-Jones, Katherine Lee, Kathy Yu, Katie Millican, Lars Lowe Sjoesund, Lisa Lee, Lucas Dixon, Machel Reid, Maciej Mikuła, Mateo Wirth, Michael Sharman, Nikolai Chinaev, Nithum Thain, Olivier Bachem,

Oscar Chang, Oscar Wahltinez, Paige Bailey, Paul Michel, Petko Yotov, Pier Giuseppe Sessa, Rahma Chaabouni, Ramona Comanescu, Reena Jana, Rohan Anil, Ross McIlroy, Ruibo Liu, Ryan Mullins, Samuel L Smith, Sebastian Borgeaud, Sertan Girgin, Sholto Douglas, Shree Pandya, Siamak Shakeri, Soham De, Ted Klimenko, Tom Hennigan, Vlad Feinberg, Wojciech Stokowiec, Yu hui Chen, Zafarali Ahmed, Zhitao Gong, Tris Warkentin, Ludovic Peran, Minh Giang, Clément Farabet, Oriol Vinyals, Jeff Dean, Koray Kavukcuoglu, Demis Hassabis, Zoubin Ghahramani, Douglas Eck, Joelle Barral, Fernando Pereira, Eli Collins, Armand Joulin, Noah Fiedel, Evan Senter, Alek Andreev, and Kathleen Kenealy. 2024. Gemma: Open models based on gemini research and technology.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models.

UCL CS TSG. 2023. Ucl department of computer science labs.

Research Computing Group UCL. 2024. Myriad.

Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. 2022. Compilable neural code generation with compiler feedback. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 9–19, Dublin, Ireland. Association for Computational Linguistics.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Huggingface's transformers: State-of-the-art natural language processing.

Cameron R. Wolfe. 2023. Language model training and inference: From concept to code. *Deci AI*.

Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2023. Large language models meet nl2code: A survey.

Qing Zhang et al. 2019. Compiler-based neural network execution: A tale of two approaches. *ACM Transactions on Computer Systems*, 37(4):1–28.