Training Code LLMs for Low-Resource and Proprietary Languages via Multi Granular Instruction Tuning and AI Feedback

Anonymous ACL submission

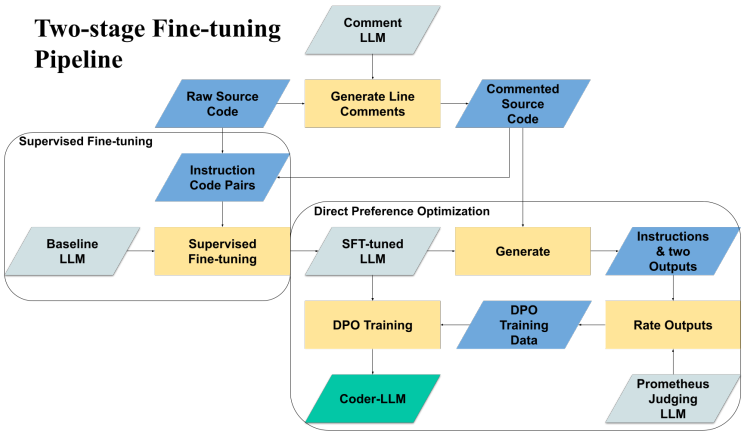

Figure 1: Illustration of the end-to-end training pipeline. Raw source code is commented, then multi-grained descriptions are generated, which serve as instructions. An LLM is fine-tuned with the multi-grained instructions. New instructions are generated using the multi-grained instructions as few-shot examples. The fine-tuned LLM generates two outputs per new instruction, and a preference is chosen by the Prometheus LLM. Finally, direct preference optimization continues training of the fine-tuned LLM using the generated preferences.

Abstract

Large Language Models (LLMs) have significantly advanced automatic code generation in mainstream programming languages but underperform on low-resource, proprietary Domain Specific Languages (DSLs) due to limited training data. We introduce a fully automated, two-stage training pipeline for coding LLMs. Stage 1 performs supervised fine-tuning (SFT) using multi granular instructions synthesized directly from raw code and comments. Stage 2 performs reinforcement learning with AI feedback (RLAIF) by generating new instructions, and subsequently performing direct preference optimization (DPO) by choosing a preferred output. We evaluate our approach using parser success, and similarity metrics, complemented with an LLM-as-a-judge approach. Our results suggest that this pipeline can enable effective adaptation of LLMs to low-resource and proprietary DSLs under realistic data and evaluation constraints.

1 Introduction

Large Language Models (LLMs) for coding have advanced program generation, synthesis, completion, and translation in mainstream programming languages such as Python and Rust (Jiang et al., 2024). However, prior work (Cassano et al., 2024; Lamas et al., 2024a; Jain et al., 2023) reports challenges when transferring to less common, low-resource, and proprietary Domain Specific Languages (DSLs). In these settings, training data in the form of instruction-code pairs is typically scarce as manual labeling is time-consuming, and functional testbeds or execution environments for validating generated code may be unavailable. This creates a gap between (i) the ever-increasing capabilities of LLMs in coding assistance on high-resource programming languages with large scale amount of training data and tooling, and (ii) low-resource, proprietary languages, where labeled data and evaluation infrastructure are limited.

To address this gap, in this paper, we introduce an automated, lightweight, two-stage training pipeline that adapts chat-based code LLMs to low-resource and proprietary languages using only raw source files, eliminating the need for functional tests, testing toolchains, execution environments, or compiler feedback during training. We use the Qwen2.5-Coder-7B-Instruct model (Yang et al., 2024) as the baseline model. In Stage 1, an LLM generates line-level comments and synthesizes multi granular natural language descriptions

of the commented code, following the approach described in (Zhang et al., 2024), which serve as instructions for training the model via supervised fine-tuning (SFT). In Stage 2 of our pipeline, we apply Reinforcement Learning with AI Feedback (RLAIF) (Lee et al., 2023) through synthesizing new instructions via few-shot prompting from the multi granular pool of code descriptions, generate two candidate code solutions with the fine-tuned model, obtain preferences from the Prometheus 2 judging LLM (Kim et al., 2024b) and optimize the model on these preferences with Direct Preference Optimization (DPO) (Rafailov et al., 2023). We further define rubric-based preference criteria optimized for evaluating code generated in arbitrary programming languages via the *Prometheus 2 LLM* (see Appendix D).

Our aim is applicability in low-resource and proprietary language settings rather than gains on established benchmarks in high-resource languages. We evaluate our approach on three general-purpose languages, starting from high- and mid-resource general-purpose programming languages (i.e., Python, Go, Rust), and on a proprietary, out-of-distribution DSL (*OOD-DSL*) for which executable functional tests are not available. For *OOD-DSL*, we leverage a pragmatic evaluation suite combining parser success, Round-Trip-Correctness (Allamanis et al., 2024) using chrF++ (Popović, 2017), and LLM-as-a-judge scoring (Gu et al., 2024). For the high- and mid-resource general-purpose programming languages, we report conventional code-generation metrics, i.e., HumanEval, HumanEval+, and MultiPL-E (Chen et al., 2021; Liu et al., 2024; Cassano et al., 2022). This mixed-methods protocol enables us to assess syntactic validity, observe and analyze surface-level fidelity, and instruction-following quality, even when execution-based testing is not possible.

Our results indicate that our pipeline outperforms the baseline model on the proprietary DSL (*OOD-DSL*) with respect to parsing rate and fidelity. On high- and mid-resource languages, we do not observe improvements over the baseline on standard benchmarks.

Summing up, our contributions are threefold:

- **An end-to-end, automated, lightweight, two-stage training pipeline for adapting code LLMs to low-resource, proprietary programming languages** using multi granular synthetic instructions, RLAIF, and DPO, without requiring human annotations, compilers, or execution environments.
- **RLAIF for code with rubric-guided DPO** through custom grading rubrics for the DPO step in LLM-as-a-judge approaches across low-resource programming language settings.
- **A practical evaluation protocol for proprietary DSLs** that combines parser success, similarity measures such as chrF++, and LLM-as-a-judge assessment, enabling evaluation without executable tests.

2 Methodology

Figure 1 presents an overview of our two-stage training pipeline.

Our aim is to adapt a chat-based code LLM to generate code in a proprietary, low-resource DSL, referred to as *OOD-DSL*. Here, instruction-code pairs are typically lacking (Tarassow, 2023; Joel et al., 2024), execution environments and functional tests may not be available, and the DSL is unlikely contained in pre-training data. Our approach is designed for exactly these constraints.

Our model is based on the Qwen2.5-Coder-7B-Instruct LLM (Hui et al., 2024b) and we adopt a two-stage training procedure, partially inspired by (Zhang et al., 2024). For *OOD-DSL*, we use a private dataset (525 files in total) provided by the DSL maintainer with a 90/10 train/test split on the file-level. We do not perform additional deduplication beyond this split; hence, the dataset may contain highly similar files across splits. For the high- and mid-resource languages (i.e., Python, Go, Rust), we sample 500 code files each from the "`bigcode/the-stack-smol`" datasets[1], matching file counts to control for dataset size.

For Stage 1, we use a commenting LLM $\mathcal{M}_{\text{comment}}$ (*Qwen2.5-32B-Instruct* (Yang et al., 2024)) to generate lightweight line-level code comments for unlabeled code files. From the commented files, $\mathcal{M}_{\text{comment}}$ produces three textual descriptions per code file: (i) a concise summary, (ii) a more detailed descriptive summary, and (iii) section-wise summaries concatenated to approximate a holistic file-level description. They serve as *instructions* for the training; the original, uncommented code is the ground truth. The resulting supervised fine-tuning dataset is denoted as $\mathcal{D}_{\text{sft}}$. For Stage 2, we apply parameter-efficient

[1] https://huggingface.co/datasets/bigcode/the-stack-smol

fine-tuning (Mangrulkar et al., 2022) with QLoRA (Dettmers et al., 2023) to impart the baseline model with an initial grasp of programming language syntax, yielding the preliminary code LLM $\mathcal{M}_{\text{sft}}$. Although this code LLM can occasionally produce syntactically valid outputs, the limited data can lead to overfitting, resulting in limited robustness and insufficient semantic understanding of the code. To improve robustness, we perform RLAIF. We first generate *novel instructions* using the commenting LLM $\mathcal{M}_{\text{comment}}$ via few-shot prompting. Specifically, given two randomly sampled *instructions* from $\mathcal{D}_{\text{sft}}$, $\mathcal{M}_{\text{comment}}$ is prompted to synthesize a new instruction. The new instruction serves two purposes: (i) the model outputs will be more varied, because we did not fine-tune on ground truth output, and (ii) the model sees a new scenario not covered during fine-tuning. For each new synthesized instruction, the fine-tuned model $\mathcal{M}_{\text{sft}}$ generates two candidate code outputs. A preference model, *Prometheus 2 LLM* $\mathcal{M}_{\text{preference}}$ (Kim et al., 2024b), evaluates the two outputs using five different grading rubrics described in Table 1. Appendix D shows the full grading rubrics in the format of a prompt on a 1-5 Likert scale. The output, which is rated better in the majority of rubrics, is labeled *chosen*, the other *rejected*. We then optimize with DPO (Rafailov et al., 2023) on *instruction*, *chosen* and *rejected* outputs to obtain the final model $\mathcal{M}_{\text{dpo}}$.

All code is licensed under the Apache 2.0 License. Hardware description, libraries, fine-tuning parameters, and GPU hours for training and evaluation are reported in Appendix A.

3 Experiments

We evaluate our two-stage training pipeline on programming languages of varying popularity: Python, Go, Rust, and *OOD-DSL*, a DSL that was (highly likely) not part of the models' pre-training data.

Different programming languages have different evaluation benchmarks, i.e., HumanEval (Chen et al., 2021) is limited to Python, whereas *OOD-DSL* lacks functional tests or reference solutions. Hence, we use standard benchmarks for high- and mid-resource languages, and unsupervised Round-Trip-Correctness (Allamanis et al., 2024) for *OOD-DSL*, plus LLM-as-a-judge (Gu et al., 2024).

Grading Rubric	**Description**
Correct	With the limited knowledge the LLM has about the DSL, choose the one that it considers more correct.
Formatted	Generated code is rarely used without human supervision. Well-formatted code is easier to read, while for some programming languages like Python, formatting can actually have semantic meaning.
Naming	Good naming of functions and variables can also help developers keep track of what variables do.
Instruction-following	It is important that the LLM actually provides a solution for the instruction, and doesn't just write good but unrelated code.
Humble	It has been shown that LLM-as-a-judge methodologies can be tricked by the model by performing self-praising at the end of the generation (Sun et al., 2023). This rubric is used to penalize such behavior.

Table 1: Evaluation criteria for assessing LLM-generated DSL code.

3.1 Comment Quality

We evaluate the quality of the synthesized code comments using six LLMs, i.e., Qwen, CodeLlama, Phi-3, Deepseek V2, WizardCoder, and Starcoder2; full model names and sizes in Appendix B. The system prompt used to evaluate the comments is given in Appendix C.

For each programming language, we evaluate 500 commented code files and focus on the quality of the most fine-grained comments, i.e., the concatenated section-wise summaries. Comments are rated on a 1-7 Likert scale (Joshi et al., 2015), with a short rationale behind the rating. We report median, minimum, and maximum values and perform a Kruskall-Wallis H-test (Kruskal and Wallis, 1952; McKight and Najab, 2010). We also qualitatively analyze generated rationales to identify differences between models.

3.2 DPO and Grading Rubrics Positional Bias

To test for positional bias in the preference model (i.e., *Prometheus 2 LLM*), we isolate it and re-run judgments with candidate outputs swapped.

We track rubric-level cases where swapping alters the preference and aggregate swap-induced reversals across the whole dataset, giving us an estimate of position bias.

3.3 DSL Evaluation via Instruction-Based Code Generation

For the test split of the *OOD-DSL*, we synthesize code descriptions to create instructions for the new code using the commenting LLM $\mathcal{M}_{\text{comment}}$.

The trained *OOD-DSL* model then generates corresponding code, which we evaluate with a proprietary parser for syntactical correctness. Note that we use this parser exclusively to evaluate our approach; it is not used in our two-stage training pipeline.

Once the generated code is parsed, we record parsing success/failure. While parsing and errors tell us about syntactic correctness, similarity to the original code is measured with the chrF++ similarity score (Popović, 2017). This evaluation protocol lets us assess syntactic correctness, as well as a similarity to the reference solution, offering both robustness in a binary correctness check, as well as a fidelity measure.

3.4 DSL Evaluation with LLM-as-a-judge

Since the DSL lacks reference test cases, we use an LLM-as-a-judge approach to estimate the quality of the generated code. We use the *Prometheus 2 LLM* direct assessment prompt (Kim et al., 2024b) with the models from Table 6, in addition to the *Prometheus 2 LLM*, on the code generated by two different models in Section 3.3, i.e., the $\mathcal{M}_{\text{sft}}$ model and the $\mathcal{M}_{\text{dpo}}$ model. We provide the original code, from which the generated code stems, as a reference in the prompt. This enables the LLMs to judge the code, even if the programming language is unknown to the LLM.

We reuse the same rubrics as in the RLAIF step of our method (see Appendix D). This allows us to evaluate the success of our method, with regard to the rubrics. Additionally, the use of multiple models, which were not used during training, ensures that the higher ratings are not due to overfitting on the *Prometheus 2 LLM* judgment.

Since the *OOD-DSL* dataset may contain highly similar source files across splits, we interpret these metrics as relative comparisons between trainings rather than as measures of generalization to unseen DSL code.

We use constrained decoding (Willard and Louf, 2023; Dong et al., 2024) to ensure the output contains a valid rating. This can, in rare cases, lead to the feedback being cut short to ensure that a rating is part of the response.

3.5 Conventional Code Generation Benchmarks

To evaluate whether our approach also benefits in well-known languages with well-established benchmarks and tests, we evaluate our approach on HumanEval, HumanEval+ (Chen et al., 2021), and MultiPL-E (Cassano et al., 2022).

For each benchmark, we evaluate the baseline model, the $\mathcal{M}_{\text{sft}}$ model, and the $\mathcal{M}_{\text{dpo}}$ model, running unit tests, and reporting pass@k, where $k = 1$, where applicable, with a greedy evaluation.

4 Results

We report results for each experiment.

4.1 Comment Quality

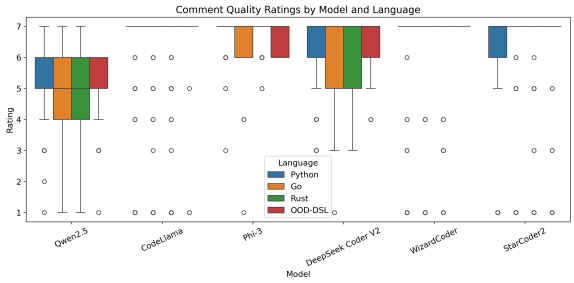

Figure 2: Boxplot ratings of LLM comment evaluation across four different languages and six different judging models.

We evaluate the quality of block comments for four programming languages using six LLM judges. Invalid LLM ratings, i.e., responses which do not match the expected output format, are removed without replacement. Figure 2 shows the rating distribution across different languages and models.

To investigate the rating behavior of different LLMs, we use a Kruskall-Wallis H Test (see Table 2), and find highly significant ($p \ll 0.001$) differences between languages. This reflects systematic variations between the models within a language.

Table 2: Kruskall–Wallis (KW) tests per language. Bold p indicates statistical significance.

Language	H	ϵ^2	p
go	899.342	0.406	$\mathbf{3.694 \times 10^{-192}}$
python	820.477	0.344	$\mathbf{4.297 \times 10^{-175}}$
rust	745.805	0.347	$\mathbf{6.110 \times 10^{-159}}$
OOD-DSL	1282.618	0.636	$\mathbf{3.723 \times 10^{-275}}$

For example, while Starcoder2 and WizardCoder primarily give the best rating, 7, Qwen and

Model	Parsing Success (%)	chrF++
SFT ($\mathcal{M}_{\text{sft}}$)	38%	42.81
DPO ($\mathcal{M}_{\text{dpo}}$)	44%	42.88

Table 3: Parsing Success and chrF++ Similarity for three models evaluated on the *OOD-DSL*.

DeepSeek-Coder are more negatively biased, with a lower median rating, but also more diverse ratings.

Additionally, some LLMs frequently respond with invalid ratings, e.g., using the wrong output format, or simply outputting an empty string, blocking us from extracting the ratings. We also see more invalid responses for low-resource languages (see Appendix E).

A qualitative analysis indicated that the rationales generated with the Likert-scale ratings correspond to each other. We found no mismatches between the ratings and the rationales. Even though some models were more diverse in phrasing the rationale, all models showed a similar understanding of the ordinal ratings, and provided similar content in a Likert-scale-rating and rationale pair.

4.2 DPO and Grading Rubrics Positional Bias

We investigate the positional bias for our grading rubrics by counting how often a swap of choices leads to the LLM picking opposite options.

We visualize the positional biases in Figure 3. We see that for each rubric, a third of decisions are not consistent when swapping the order of choices. We also see that for some responses the choice could not be extracted, this happens most frequently for the "Correct" rubric.

4.3 DSL Evaluation via Instruction-Based Code Generation

In our experiments, the baseline model produced invalid code in this DSL setting, which we attribute to the DSL not being contained in its training data. The initial fine-tuned model M_{sft} achieves a parsing success of 38%, and a ChrF++ similarity score of 42.81. Finally, the fully trained model M_{dpo} achieves consistently higher parsing success of 44% than M_{sft} under the same evaluation conditions.

Table 3 reports the parsing rate, and the chrf++ similarity metric between the $\mathcal{M}_{\text{sft}}$ model and the $\mathcal{M}_{\text{dpo}}$ model.

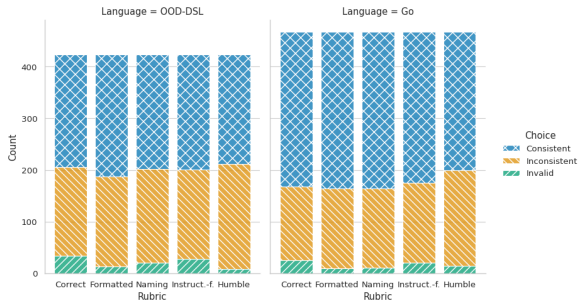

Figure 3: Choices of *Prometheus 2 LLM* in a low-resource (*OOD-DSL*) and a mid- to high-resource programming language setting (Go) for our rubrics. We visualize when the LLM chooses the same output consistently (`Consistent`), even when the choices are swapped in the input. Otherwise, we consider the choice inconsistent (`Inconsistent`), which hints at a positional bias or random guessing. Finally, we show the number of invalid choices (`Invalid`), where at least one of the choices could not be extracted from the response. We only visualize the train split of *OOD-DSL*. When computing the difference between consistent and inconsistent choices for *OOD-DSL*, we observe a difference of 8 for the Humble rubric, and 39 to 62 for the rest of the rubrics. For Go the choices are more consistent, with the difference ranging from 81 (again the Humble rubric) to 156, presumably because Go was in the models' pre-training data.

4.4 DSL Evaluation with LLM-as-a-judge

We also evaluate the generated code in the previous section using an LLM-as-a-judge approach described in Section 3.4. Table 4 contains the ratings, per rubric, of 8 open-weight models (detailed in Table 7) for the $\mathcal{M}_{\text{sft}}$ model and the $\mathcal{M}_{\text{dpo}}$ model.

Most models assign higher scores to the *DPO* model for the majority of rubrics. The exceptions are the two smallest models. For *DeepSeek Coder* the models are tied, while *Prometheus v2* gives the *SFT* model slightly higher scores for 4 rubrics.

Only 5 rating differences in Table 4 are statistically significant at $\alpha = 0.1$ (without multiple-comparison correction, e.g., Bonferroni). We attribute this to the small number of test examples (≈ 50) and the judging LLM not knowing the programming language.

4.5 Code Generation Benchmarks

These metrics stay stable during training, both in the $\mathcal{M}_{\text{sft}}$ model, and the $\mathcal{M}_{\text{dpo}}$ model. We run the evaluations with the Bigcode evaluation harness (Ben Allal et al., 2022). Since we train instruction LLMs, we adapt the MultiPL-E evaluations for Go and Rust to Qwen's provided "chat tem-

Model	Rubric				
	Correct	Formatted	Naming	Instruction-f.	Humble
DeepSeek Coder (6.7B)					
SFT	2.92	3.50	3.85	**3.95**	**3.89**
DPO	**3.33**	**4.00**	3.85	3.88	3.50
Prometheus v2 (7B)					
SFT	**2.66**	**3.26**	**3.18**	2.62	**3.24**
DPO	2.62	3.20	3.12	**2.96**	3.12
Llama 3.1 (8B)					
SFT	**3.50**	**3.81**	3.68	3.72	3.05
DPO	3.29	3.72	**3.72**	**4.00**	**3.38**
CodeLlama (13B)					
SFT	3.06	3.62	**3.64**	3.42	3.40
DPO	**3.21**	**3.88**	3.36	**3.54**	**3.81 .**
Qwen 2.5 (14B)					
SFT	2.58	3.79	3.64	3.03	3.47
DPO	**2.89 .**	**3.81**	**3.72**	**3.20**	**3.53**
DeepSeek Coder V2 Lite (16B)					
SFT	2.80	3.51	3.70	3.77	**3.78**
DPO	**2.86**	**4.04 .**	**3.98**	**3.96**	3.75
GPT-OSS (20B)					
SFT	1.72	**2.00**	3.90	3.31	3.00
DPO	**2.04**	1.63	**4.18**	**3.77**	**3.11**
Gemma 3 (27B)					
SFT	2.78	4.00	3.38	3.00	3.00
DPO	**3.13 .**	4.00	**3.40**	**3.50**	3.00
Global Mean Rating					
SFT	2.77	3.52	3.59	3.32	3.38
DPO	**2.88**	**3.66**	**3.63**	**3.54 .**	**3.45**

Table 4: Mean ratings (rounded to 2 decimals) for *SFT* and *DPO* models under five rubrics. The higher rating within each model and rubric is **bold**. We mark statistically significant differences with **.** ($\alpha = 0.1$), using a Mann-Whitney U test.

plate". As evident from the results, in Table 5, the performance for the language that is trained does not change much. Although, we do not notice any strong improvements, the size and granularity of these benchmarks are not detailed enough to give us insights into small behavioral changes our two-stage training procedure might create in the models.

Table 5: Evaluation on Code Benchmark.

Train Stage	Language	Instruct HumanEval	HumanEval+	MultiPL-Go	MultiPL-Rust
Baseline Model	–	0.848	0.195	0.608	0.696
SFT ($\mathcal{M}_{\text{sft}}$)	Python	0.753	0.116	0.595	0.627
	Go	0.780	0.226	0.608	0.671
	Rust	0.750	0.140	0.608	0.658
DPO ($\mathcal{M}_{\text{dpo}}$)	Python	0.823	0.091	0.588	0.677
	Go	0.811	0.152	0.601	0.709
	Rust	0.811	0.171	0.589	0.728

5 Discussion

5.1 Application of Method

Our method is especially efficient in settings of low-resource and proprietary languages, where the syntax and semantics of the language are not previously known to the model, as well as settings where traditional evaluation pipelines are infeasible. In the case of the proprietary, low resource, *OOD-DSL*, our method enables systematic evaluation without explicit access to testing toolchains or functional testcases. Through a combination of parser feedback and the chrF++ fidelity metric, we measured both binary correctness of the implementation, as well as code similarity. Similarly, applying LLM-as-a-judge further enriches the evaluation in settings where resources are sparsely available.

5.2 Generalizability of Method

Although we observed no improvements across general-purpose (i.e. Python, Go, Rust), potential improvements could be expected in settings where the new data used for training is substantial in quantity and heterogeneity. We identify several reasons for the comparatively same results as with the baseline model, namely i) most of the data and syntax that is used for fine-tuning is probably already identically, or similarly, in the training set of the model, and ii) there is no significant new data provided to the model in the fine-tuning process, thus leading to no statistically significant difference in model performance and accuracy.

5.3 Reliability of LLM-as-a-Judge

A critical finding in our evaluation is the systematic variability in LLM-as-a-judge ratings. Across all languages and the comments generated for the code blocks, we observe highly significant between-language differences (i.e., Kruskall-Wallis $\ll 0.001$). Moreover, a lower mean/median score does not necessarily reflect a worse judging model, but rather a more strict internal evaluation guideline between the models. Consequently, any deployment of LLM-as-a-judge would require clear constraints and prompts across judging models in order to mitigate systematic biases.

5.4 Bias in Preference Data

The analysis of the *Prometheus 2 LLM* further revealed positional sensitivity in preference judgment in A/B positional bias. Swapping the order of candidate A/B outputs led to measurable reversal in rubric-level preferences, suggesting that dataset construction can have an impact on the preference of grading rubric choices. Although the absolute magnitude of the bias was not severe, it still had an impact on the training process, thus raising the need for mitigating this bias. This introduces a necessity

for randomized presentation orders in preference data in order to increase robustness in the DPO training process.

5.5 Broader Implications

Taken together, these results suggest two key takeaways: Firstly, our approach provides a viable and efficient approach in adapting chat-based code LLMs to low-resource and proprietary languages under realistic data and evaluation constraints. Secondly, while LLMs can serve as judges, their bias, validity, and positional preference has to be considered if such approaches are to be applied for evaluation. Therefore, it is important to consider validity rates, employ calibration of judge scores, as well as explore methods such as multi-judge aggregation of scores (majority voting). These measures are quintessential in ensuring alignment and robustness of the approach to reflect genuine quality differences, rather than effects of judge model idiosyncrasies.

6 Related Work

There have been multiple advances in coding LLMs, such as the Qwen2.5 and Qwen3 family of models (Yang et al., 2025, 2024), Deepseek (Guo et al., 2024; Zhu et al., 2024; Shojaee et al., 2023) showing good performance and quality of output across different benchmarks such as HumanEval, HumanEval+, MBPP, and Multipl-E (Yu et al., 2024; Liu et al., 2024; Chen et al., 2021; Cassano et al., 2022).

The underlying methodologies used to fine-tune these methods stem from older models. Phi-1 (Gunasekar et al., 2023) showed that synthesizing high-quality code tasks from a larger LLM ($>$ 70B parameters), allows fine-tuning highly-capable small models (1.3B parameters). Similarly, Code Alpaca (Chaudhary, 2023) synthesized instruction-following data, adapting Self-Instruct (Wang et al., 2023b) to code generation. Furthermore, WizardCoder (Luo et al., 2024) synthesized an instruction fine-tuning dataset, adapting the Evol-Instruct (Xu et al., 2024) method to *evolve* existing instruction data from Code Alpaca (Chaudhary, 2023). This method requires a seed dataset with existing instruction-output pairs.

Often times, coding LLMs were further aligned with Direct Preference Optimization (DPO) (Rafailov et al., 2023), and DPO with an offset (Amini et al., 2024). One such approach was used to create Qwen2.5-Coder (Hui et al., 2024a). To this end, multiple generations are sampled and feedback is provided, consisting of parsing errors and synthetic unit tests. Based on the unit test, or using an LLM-as-a-judge (Zheng et al., 2023b), then provides a final preference. This approach requires both parsing and an execution environment for the programming languages to learn, further limited by the process of using LLM-as-a-judge for code being understudied (Gu et al., 2024). These LLM-as-a-judge are oftentimes reinforced with judging-LLMs specifically trained for such tasks such as Prometheus and Prometheus 2 (Kim et al., 2023, 2024b; Lee et al., 2024; Kim et al., 2024a). However, these approaches are also oftentimes not directly optimized for code evaluations, as the rubrics are designed for a more general-purpose assessment.

One disadvantage is that all of these methods require functional tests, testing toolchains, and execution environments. There are several approaches which synthesize instructions and create preference datasets, most notably: i) Magicoder (Wei et al., 2024b), which constructs instruction datasets for coding using OSS-Instruct, generating problems and solutions for open source code snippets, ii) OctoPack (Muennighoff et al., 2024), which creates an instruction tuning dataset, scraping commits from GitHub and using the commit message as the instruction, and the code pre-commit as input, and post-commit as output, and iii) Dolphcoder (Wang et al., 2024), which is trained in two stages. First, outputs for an instruction dataset, created similarly to WizardCoder (Luo et al., 2024), are generated by GPT3.5, and a baseline model is fine-tuned on this dataset. Next, the fine-tuned model generates code multiple times for some of the instructions, and GPT4 provides evaluation feedback. The fine-tuned model is further fine-tuned, given code as input to generate evaluation feedback. Similar to our approach, training is split up into two stages, but while our approach only requires raw code of the target programming language, Dolphcoder needs instructions for generating code in a target programming language in addition to an existing pre-trained LLM that is already proficient in the target programming language.

Other approaches have shown that the grammar of DSLs can be used to enable LLMs to write syntactically correct grammar (Wang et al., 2023a). This approach adds the DSL grammar to the prompt, enabling the LLM to create valid DSL

files frequently. This, in combination with fine-tuning and few-shot prompting, is used by DSL-Xpert (Lamas et al., 2024b), enabling the creation of DSL coding agents. We view this method as orthogonal to ours, and our trained models can be used with grammar prompting during inference. Furthermore, this approach requires access to the grammar, typically in a specific format, which might not be the case for DSLs created in proprietary or niche frameworks.

To evaluate these approaches in low-resource and proprietary language settings, conventional testing methodologies can oftentimes not be applied, as functional tests, testing toolchains, and execution environments are missing. Hence, evaluations rely on word-based, character-based, or token-based similarity metrics such as chrF/chrF++ (Popović, 2015, 2017), to evaluate code quality (Evtikhiev et al., 2023). Since instruction-solution pairs are rare here, unsupervised Round-Trip-Correctness (Allamanis et al., 2024) was developed. Finally, LLM-as-a-judge uses LLMs directly to evaluate generations based on prompts (Li et al., 2024; Zheng et al., 2023a; Gu et al., 2024; Wei et al., 2024a).

7 Conclusion

In this work, we introduced a two-stage training pipeline for adapting chat-based code LLMs to low-resource and proprietary programming languages without functional tests, testing toolchains, and execution environments. Our method combines multi granular synthetic instructions generated by a commenting LLM with RLAIF, and further incorporates preference A/B alignment through DPO.

Through experiments across three general-purpose programming languages (Python, Go, Rust), and one proprietary, *OOD-DSL* language, we have shown that our approach is well-applicable in low-resource and proprietary language settings, where the LLM has likely not seen the language during the trainings. Our evaluation pipeline for the DSL, combining parser success/failure, chrF++ similarity, as well as LLM-as-a-judge showed that our two-stage training pipeline outperforms the baseline models in syntactic and semantic understanding of the programming language, making our approach viable in such settings.

Beyond task performance, our approach also revealed critical considerations for LLM-as-a-judge approaches when it comes to between-judge variations. We observed substantial variations between model judges, frequent invalid outputs between different judging models, as well as moderate positional bias in the DPO preference alignment. These findings emphasize the importance of careful normalization of prompts, as well as a conscientious mitigation of positional bias by applying methods such as random positioning or majority judge-voting through multiple grading iterations.

Overall, our study highlights both the opportunities and challenges of adopting code LLMs to low-resource and proprietary language settings. We show that multi granular instructions tuning and AI-driven feedback through RLAIF can increase the model correctness. We hope this work encourages research on robust adaptation of pipelines for DSL fine-tuning in low-resource and proprietary language settings, while also broadening the applicability of local code LLMs in settings which go beyond "mainstream" benchmarks and languages.

8 Future Work

Executable Evaluation for DSLs: The next step in elevating the quality of the evaluation is to expand the functionality of the *OOD-DSL* with executable testbeds. In this way, task suites and unit tests can be incorporated into the workflow in order to quantitatively measure the quality of the control flow, as well as apply more functional correctness testing like pass@k.

Richer Alignment Signals: In alignment, we intend to i) scale multi granular instruction generation on more code base retrievals to increase its scope, ii) study additional preference objectives, and iii) apply iterative reinforcement learning in execution time to create a tradeoff of quality of output and inquiry time, depending on the GPU power.

Judge Calibration and Reliability: Given the between-judge variability in the comment quality evaluation, and in the LLM-as-a-judge, we intend to i) calibrate judges per language through score normalization, ii) implement between-judge agreements and majority voting, rather than report individual judge results, and iii) randomize presentation order to mitigate positional bias.

9 Ethical Concerns

Safety in Real Systems: If a DSL ultimately controls real devices, for example engines, or devices which can set a certain temperature, pressure, etc.,

these devices might become unsafe to use, or the code generated by the LLM coder can lead to the devices getting damaged. This can also lead to injuries in case that humans physically operate such devices.

Over-reliance: Models aligned with AI-in-the-loop approaches are trained on data. It is important to not be over-reliant on the support tool and always fall back onto documentation, safety measures, failsafes, and supported scenarios.

Privacy: If log data or certain prompts which are documented contain personal data or personal identifiers, we need to apply redaction and minimization to filter out this data, eliminating the traces of personal identifiers in the raw files.

Evaluation Bias and Fairness: LLM-as-a-judge can introduce evaluation bias, as different LLMs are trained with different data. To mitigate this, it is critical to randomly order data, perform majority judging and voting, and ensure that the training data is objective and diversified.

Data Governance and Intellectual Property: When adapting proprietary languages, training data might contain confidential or copyrighted information. Moreover, the proprietary language in itself might be confidential in itself. To combat this, everything should be ran locally, and confidential identifiers should be further screened and removed as a failsafe against intellectual property violation.

10 Declaration of generative AI and AI-assisted technologies in writing

During the preparation of this work, the author(s) used LanguageTool, ChatGPT, Claude, and Le Chat to : perform grammar and spelling checks, improve writing style, paraphrase and reword, and format tables. After using these tools, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

11 Limitations

Lack of reference tests for the DSL: Our primary DSL evaluation relies on parser success, chrF++ similarity, as well as an LLM-as-a-judge approach. In the absence of executable reference unit tests, the applied methods might not fully capture the functional correctness of the code, as well as the efficiency of the implementation. Moreover, the quantity of the tests in the given use-case was very limited, hence potentially affecting the results reported.

Judge variability: Conclusions that depend on LLM-as-a-judge approach are sensitive to LLM biases, as well as prompt definitions, which might highly impact the output of the LLM-as-a-judge pipeline. Although we try to mitigate this as much as possible, residual bias may persist due to model training data bias which is unknown to us, as well as flaws in the prompts given, and the way different models might perceive it.

Generalizability to high-resource languages: Even though our method was shown to be efficient on low-resource and proprietary languages, our method showed modest to no improvements on conventional benchmarks for high-resource programming languages. This makes our method *not efficient* in beating SOTA models in high-resource settings.

Compute and data scope: Although parameter-efficient tuning keeps the training costs low, our experiments, and subsequently our method, are bound by available datasets and hardware. Larger data or model scopes could change quantitative outcomes, and might require a stronger hardware setup to be ran, which is not always available.

Metrics scope: Tests such as chrF++ capture lexical and structural similarity, but do not offer a a deeper semantic or performance analysis; likewise, parser success does not guarantee runtime safety, nor provide a good measure of overall correctness. Because of this, it is important to refer to Future Work (see Section 8) to expand the testing framework through unit tests and functional tests to ensure solution robustness.

References

Miltiadis Allamanis, Sheena Panthaplackel, and Pengcheng Yin. 2024. Unsupervised evaluation of code llms with round-trip correctness. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Afra Amini, Tim Vieira, and Ryan Cotterell. 2024. Direct preference optimization with an offset. *arXiv preprint arXiv:2402.10571*.

Loubna Ben Allal, Niklas Muennighoff, Logesh Kumar Umapathi, Ben Lipkin, and Leandro von Werra. 2022. A framework

for the evaluation of code generation models. https://github.com/bigcode-project/bigcode-evaluation-harness.

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge transfer from high-resource to low-resource programming languages for code llms. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):677–708.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *Preprint*, arXiv:2208.08227.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115.

Yixin Dong, Charlie F Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. 2024. Xgrammar: Flexible and efficient structured generation engine for large language models. *Proceedings of Machine Learning and Systems 7*.

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the BLEU: how should we assess quality of the code generation models? *J. Syst. Softw.*, 203:111741.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, and 1 others. 2024. A survey on llm-as-a-judge. *arXiv preprint arXiv:2411.15594*.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks are all you need. *CoRR*, abs/2306.11644.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024a. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024b. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Rijul Jain, Wode Ni, and Joshua Sunshine. 2023. Generating domain-specific programs for diagram authoring with large language models. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 70–71.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Sathvik Joel, Jie JW Wu, and Fatemeh H Fard. 2024. A survey on llm-based code generation for low-resource and domain-specific programming languages. *arXiv preprint arXiv:2410.03981*.

Ankur Joshi, Saket Kale, Satish Chandel, and D Kumar Pal. 2015. Likert scale: Explored and explained. *British journal of applied science & technology*, 7(4):396.

Seungone Kim, Jamin Shin, Yejin Cho, Joel Jang, Shayne Longpre, Hwaran Lee, Sangdoo Yun, Seongjin Shin, Sungdong Kim, James Thorne, and 1 others. 2023. Prometheus: Inducing fine-grained evaluation capability in language models. *arXiv preprint arXiv:2310.08491*.

Seungone Kim, Juyoung Suk, Ji Yong Cho, Shayne Longpre, Chaeeun Kim, Dongkeun Yoon, Guijin Son, Yejin Cho, Sheikh Shafayat, Jinheon Baek, Sue Hyun Park, Hyeonbin Hwang, Jinkyung Jo, Hyowon Cho, Haebin Shin, Seongyun Lee, Hanseok Oh, Noah Lee, Namgyu Ho, and 13 others. 2024a. The biggen bench: A principled benchmark for fine-grained evaluation of language models with language models. *Preprint*, arXiv:2406.05761.

Seungone Kim, Juyoung Suk, Shayne Longpre, Bill Yuchen Lin, Jamin Shin, Sean Welleck, Graham Neubig, Moontae Lee, Kyungjae Lee, and Minjoon Seo. 2024b. Prometheus 2: An open source language model specialized in evaluating other language models. *arXiv preprint arXiv:2405.01535*.

William H Kruskal and W Allen Wallis. 1952. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621.

Victor Lamas, Miguel R. Luaces, and Daniel Garcia-Gonzalez. 2024a. Dsl-xpert: Llm-driven generic DSL code generation. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, MODELS Companion 2024, Linz, Austria, September 22-27, 2024*, pages 16–20. ACM.

Victor Lamas, Miguel R. Luaces, and Daniel Garcia-Gonzalez. 2024b. Dsl-xpert: Llm-driven generic dsl code generation. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, MODELS Companion '24, page 16–20, New York, NY, USA. Association for Computing Machinery.

Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Ren Lu, Thomas Mesnard, Johan Ferret, Colton Bishop, Ethan Hall, Victor Carbune, and Abhinav Rastogi. 2023. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *OpenReview preprint*.

Seongyun Lee, Seungone Kim, Sue Hyun Park, Geewook Kim, and Minjoon Seo. 2024. Prometheus-vision: Vision-language model as a judge for fine-grained evaluation. *Preprint*, arXiv:2401.06591.

Haitao Li, Qian Dong, Junjie Chen, Huixue Su, Yujia Zhou, Qingyao Ai, Ziyi Ye, and Yiqun Liu. 2024. Llms-as-judges: a comprehensive survey on llm-based evaluation methods. *arXiv preprint arXiv:2412.05579*.

Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. 2024. Evaluating language models for efficient code generation. In *First Conference on Language Modeling*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. Peft: State-of-the-art parameter-efficient fine-tuning methods. `https://github.com/huggingface/peft`.

Patrick E McKight and Julius Najab. 2010. Kruskal-wallis test. *The corsini encyclopedia of psychology*, pages 1–1.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Maja Popović. 2015. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, Lisbon, Portugal. Association for Computational Linguistics.

Maja Popović. 2017. chrf++: words helping character n-grams. In *Proceedings of the second conference on machine translation*, pages 612–618.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

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

Zhiqing Sun, Yikang Shen, Hongxin Zhang, Qinhong Zhou, Zhenfang Chen, David Cox, Yiming Yang, and Chuang Gan. 2023. Salmon: Self-alignment with principle-following reward models. *Preprint*, arXiv:2310.05910.

Artur Tarassow. 2023. The potential of llms for coding with low-resource and domain-specific programming languages. *arXiv preprint arXiv:2307.13018*.

Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. 2023a. Grammar prompting for domain-specific language generation with large language models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Yejie Wang, Keqing He, Guanting Dong, Pei Wang, Weihao Zeng, Muxi Diao, Weiran Xu, Jingang Wang, Mengdi Zhang, and Xunliang Cai. 2024. DolphCoder: Echo-Locating Code Large Language Models with Diverse and Multi-Objective Instruction Tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4706–4721, Bangkok, Thailand. Association for Computational Linguistics.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023b. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 13484–13508. Association for Computational Linguistics.

Hui Wei, Shenghua He, Tian Xia, Fei Liu, Andy Wong, Jingyang Lin, and Mei Han. 2024a. Systematic evaluation of llm-as-a-judge in llm alignment tasks: Explainable metrics and diverse prompt templates. *arXiv preprint arXiv:2408.13006*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024b. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Brandon T Willard and Rémi Louf. 2023. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, and 22 others. 2024. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. 2024. Humaneval pro and mbpp pro: Evaluating large language models on self-invoking code generation. *arXiv preprint arXiv:2412.21199*.

Yongan Zhang, Zhongzhi Yu, Yonggan Fu, Cheng Wan, and Yingyan Celine Lin. 2024. MG-Verilog: Multi-grained Dataset Towards Enhanced LLM-assisted Verilog Generation. *arXiv preprint*. ArXiv:2407.01910 [cs].

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, and 1 others. 2023a. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023b. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

A Training setup

Table 8 describes the hardware specifications used for fine-tuning and evaluation.

Table 9 describes the estimated GPU time needed for the training and the evaluation.

Table 10 describes the libraries used and their versions.

The code block in Figure 4 shows the hyperparameters of the Q-LoRA fine-tuning in Stage 1 (SFT).

The code block in Figure 5 shows the hyperparameters of the Q-LoRA fine-tuning in Stage 2 (DPO).

B Models Used for Evaluating

Model	Parameters
Qwen/Qwen2.5-14B-Instruct	14B
codellama/CodeLlama-13b-Instruct-hf	13B
microsoft/Phi-3-medium-128k-instruct	14B
deepseek-ai/ DeepSeek-Coder-V2-Lite-Instruct	16B
WizardLMTeam/WizardCoder-15B-V1.0	15B
bigcode/starcoder2-15b	15B

Table 6: Large Language Models (13B–16B parameters) used for benchmarking code comment quality.

Model	Parameters
deepseek-ai/deepseek-coder-6.7b-instruct	6.7B
prometheus-eval/prometheus-7b-v2.0	7B
meta-llama/Llama-3.1-8B-Instruct	8B
codellama/CodeLlama-13b-Instruct-hf	13B
Qwen/Qwen2.5-14B-Instruct	14B
deepseek-ai/ DeepSeek-Coder-V2-Lite-Instruct	16B
openai/gpt-oss-20b	20B
google/gemma-3-27b-it	27B

Table 7: Large Language Models (6.7B–27B parameters) used for rating LLM generated code.

Component	Specification
GPU	2× NVIDIA A5000 (24 GB VRAM each)
RAM	128 GB
CPU	AMD EPYC 9334 (32 Core)

Table 8: Hardware configuration used for model evaluation and fine-tuning.

Stage	Per Language (h)	Total (4 Languages)
Training		
Fine-tuning per language	≈18	$4\times 18 =$ **72**
Subtotal (Training)		**≈ 72 hours**
Evaluation		
Evaluation per language	≈5	$4 \times 5 = 20$
Subtotal (Evaluation)		**≈ 20 hours**
Estimated Total GPU Hours		**≈ 92 hours**

Table 9: Estimated time required for training and evaluation across four programming languages. All times are expressed in GPU hours using 2× NVIDIA A5000 GPUs.

Library	Version
accelerate	1.3.0
axolotl	0.12.2
bitsandbytes	0.45.2
datasets	3.2.0
evaluate	0.4.1
huggingface-hub	0.29.3
numpy	1.26.4
optimum	1.16.2
pandas	2.2.3
peft	0.14.0
safetensors	0.6.1
scikit-learn	1.4.2
scipy	1.15.2
sentencepiece	0.2.0
tokenizers	0.21.4
torch	2.5.1+cu124
torchaudio	2.5.1+cu124
torchvision	0.20.1+cu124
transformers	4.48.3
triton	3.1.0
trl	0.15.0
vllm	0.10.1
wandb	0.19.8
xformers	0.0.28.post3

Table 10: Core libraries and versions used in our experiments. CUDA toolchain: 12.4; Python: 3.10.16.

```yaml
run_name: "7B-sft"

base_model:
  model_type: AutoModelForCausalLM
  tokenizer_type: AutoTokenizer

load_in_8bit: false
load_in_4bit: true
strict: false

datasets:
  - path: $dataset
    type: chat_template
    train_on_split: train
    roles_to_train: ["assistant"]
    train_on_eos: turn
    field_messages: messages

val_set_size: 0.05
sequence_len: 4096
pad_to_sequence_len: true
sample_packing: true
adapter: qlora

lora_r: 32
lora_alpha: 32
lora_dropout: 0.05

gradient_accumulation_steps: 4
micro_batch_size: 1
num_epochs: 5
optimizer: adamw_torch_fused
lr_scheduler: cosine
learning_rate: 0.00005
flash_attention: true
weight_decay: 0.1
warmup_steps: 10
```

Figure 4: Training configuration and key hyperparameters for fine-tuning.

```yaml
run_name: "7B-dpo"
base_model:
  model_type: AutoModelForCausalLM
  tokenizer_type: AutoTokenizer
load_in_8bit: false
load_in_4bit: true
strict: false
rl: dpo
datasets:
  - path: $dataset
    ds_type: json
    data_files:
      - $dataset
    type: chat_template
    train_on_split: train
    split: train
    field_messages: messages
    field_system: system
    field_chosen: chosen
    field_rejected: rejected
    message_field_role: role
    message_field_content: content
    roles:
      system:
        - system
      user:
        - user
      assistant:
        - assistant
val_set_size: 0.1
dataset_prepared_path: last_run_prepared
output_dir:
sequence_len: 8192
pad_to_sequence_len: true
sample_packing: false
adapter: qlora
lora_model_dir: "please-overwrite"
lora_r: 32
lora_alpha: 32
lora_dropout: 0.05
lora_target_linear: true
lora_fan_in_fan_out:
wandb_project: "dpo"
wandb_entity:
wandb_watch:
wandb_name: "dpo"
wandb_log_model:
gradient_accumulation_steps: 32
micro_batch_size: 1
eval_batch_size: 1
num_epochs: 5 # set to `2` for OOD-DSL, because of quicker convergence (reaches substantial above 1 and not significantly higher than epoch 4)
optimizer: adamw_torch_fused
lr_scheduler: cosine
learning_rate: 0.0002

train_on_inputs: false
group_by_length: false
bf16: auto
fp16:
tf32: false
gradient_checkpointing: true
early_stopping_patience:
resume_from_checkpoint:
local_rank:
logging_steps: 1
xformers_attention:
flash_attention: true
sdp_attention: false

loss_watchdog_threshold: 50.0
loss_watchdog_patience: 3
warmup_steps: 10
evals_per_epoch: 4
eval_table_size:
eval_max_new_tokens: 256

saves_per_epoch: 2
debug:
deepspeed:
weight_decay: 0.0
fsdp:
fsdp_config:
special_tokens:

ddp_timeout: 36000000
```

Figure 5: Direct Preference Optimization (DPO) fine-tuning configuration and key hyperparameters.

C Prompt used for evaluation of comment blocks

In order to evaluate the comment blocks and their integrity and quality, different model were used to benchmark the quality of the comments generated for the code blocks with. The following prompt was used in order to evaluate whether the comment is good or not on a 1-7 Likert Scale.

```
System:

You are a coding annotator specialized in
    ↪ instruction-tuning and AI feedback for
    ↪ code comments. You are given the task to
    ↪ evaluate the quality of a set of
    ↪ comments that will be provided to you.
    ↪ You will provide me with a single 1 to 7
    ↪ Likert-scale rating based on the
    ↪ following criteria: correctness,
    ↪ descriptiveness, readability of the
    ↪ comments.

Correctness: Do the comments adequately and
    ↪ accurately describe the functionality
    ↪ and the purpose of the code block which
    ↪ was provided to you? If the comments are
    ↪ absolutely unrelated, irrelevant, or
    ↪ incorrect to the code block, rate it as
    ↪ 1.

Descriptiveness: Do the comments provide
    ↪ sufficient details to understand the
    ↪ code block? Comments that are too vague
    ↪ or lack important information should be
    ↪ rated lower on the Likert-scale compared
    ↪ to other comments which are more
    ↪ substantive. Do not give higher ratings
    ↪ to simply longer comments, as the
    ↪ comment length should not be a valid and
    ↪ indicative factor in this rating.

Readability: Are the comments easy to read,
    ↪ comprehend, and understand? If the
    ↪ comments are cluttered, poorly worded,
    ↪ or unclear, they should be rated lower.
    ↪ This also applies to comments which are
    ↪ unnecessarily long, repetitive, or do
    ↪ not provide useful insight.

Please provide a single integer rating between
    ↪ 1 and 7 based on the quality of the
    ↪ comments:

1: The comments are completely incorrect or
    ↪ irrelevant to the code block.

7: The comments are completely accurate, very
    ↪ descriptive, and easy to understand.

The code block is as follows:
{code_block}

The comments are as follows:
{comments}

Structure your response as follows, and only
    ↪ output this once:
```

```
rating: [rating]
rationale: [rationale]
End your response with: <END>
```

D Prompt used for self-defined preference rubrics

In order to evaluate the code, we have defined five preference rubrics, and also given a 1-5 Likert scale rating for the rubrics. The prompt can be found here:

```
Correct: |-
  [Does the code run and meet the specified
      ↪ behavior?]
  Score 1: Fails to implement - Won't run or is
      ↪ mostly broken (major syntax/semantic
      ↪ errors). Task not implemented.
  Score 2: Largely incorrect - Some relevant
      ↪ pieces, but major parts missing or
      ↪ wrong. Frequent errors; mostly
      ↪ incorrect results.
  Score 3: Partially correct - Runs/compiles
      ↪ and attempts the task, but produces
      ↪ wrong results for many cases; core
      ↪ logic incomplete or flawed.
  Score 4: Mostly correct - Correct for typical
      ↪ cases, but misses some edge cases
      ↪ (e.g., empty input, extreme values,
      ↪ invalid/None). Minor bugs possible.
  Score 5: Fully correct & robust - Implements
      ↪ the spec, runs without errors, and
      ↪ handles reasonable edge cases as above.
  Notes for the grader-model: Judge functional
      ↪ behavior only (ignore style/perf
      ↪ unless it breaks correctness). If
      ↪ unsure between two scores, choose the
      ↪ lower one.

Formatted: |-
  [Is the code well-structured and easy to read
      ↪ (Ignore identifier quality/naming.)?]
  Score 1: Unreadable - Severe/mixed
      ↪ indentation errors; chaotic layout.
      ↪ Inconsistent/missing line breaks and
      ↪ spacing. Style is all over the place.
  Score 2: Poor - Generally hard to read. Some
      ↪ indentation/alignment mistakes.
      ↪ Inconsistent spacing/brace layout.
      ↪ Little organization; formatting often
      ↪ hinders understanding.
  Score 3: Adequate - Readable overall with
      ↪ noticeable issues (e.g., uneven
      ↪ spacing/wrapping or mixed styles).
      ↪ Basic structure present
      ↪ (functions/modules), but some clutter.
  Score 4: Good - Clean structure and flow.
      ↪ Consistent indentation and brace
      ↪ style. Sensible line breaks and
      ↪ spacing. Minor inconsistencies that
      ↪ don't impede reading.
  Score 5: Excellent - Polished, consistent
      ↪ formatting throughout. Idiomatic
      ↪ layout for the language. Clear
      ↪ organization
      ↪ (functions/classes/sections), helpful
      ↪ whitespace/comments, and consistent
      ↪ bracket/operator spacing.
  Notes for the grader-model: Judge
      ↪ formatting/structure only
      ↪ (indentation, whitespace, line breaks,
      ↪ brace style, organization,
      ↪ comments/docstrings). Ignore
      ↪ correctness and variable names. If
      ↪ between scores, choose the lower one.

Naming: |-
  [Are identifiers (variables, functions,
      ↪ classes, params) descriptive and
      ↪ informative?]
  Score 1: Nonsensical - Mostly single letters
      ↪ or random strings/digits (beyond
      ↪ trivial i/j/k loop indices).
      ↪ Misleading names; same name reused for
      ↪ different things.
  Score 2: Poor - Frequently vague or
      ↪ misleading. Inconsistent or
      ↪ unexplained abbreviations/suffixes.
      ↪ Noticeable reuse/shadowing that
      ↪ confuses purpose.
  Score 3: Mixed - Some clear names, but many
      ↪ are too short/long or ambiguous.
      ↪ Meaning often guessable but
      ↪ unreliable. Occasional misleading
      ↪ names or reuse.
  Score 4: Good - Mostly descriptive and
      ↪ concise. Names reflect role/behavior.
      ↪ Conventions are largely consistent.
      ↪ Minor issues (a few unclear
      ↪ abbreviations or wordy names).
  Score 5: Excellent - Clear, precise, and
      ↪ concise throughout. Names consistently
      ↪ convey intent (e.g., verbs for
      ↪ functions, nouns for data; is_/has_
      ↪ for booleans). No misleading
      ↪ reuse/shadowing.
  Notes for the grader-model: Judge naming only
      ↪ (not formatting or correctness). Don't
      ↪ penalize names mandated by an
      ↪ API/spec. Brief single-letter indices
      ↪ in tiny loops are acceptable.

"Instruction-following": |-
  [Does the submission follow the prompt's
      ↪ requirements and constraints?]
  Score 1: Noncompliant - No code or unrelated
      ↪ output; ignores key instructions
      ↪ (e.g., wrong
      ↪ language/format/signature) or includes
      ↪ prohibited content.
  Score 2: Weak - Tangential to the task.
      ↪ Implements small fragments but misses
      ↪ major requirements and/or breaks key
      ↪ constraints (wrong interface,
      ↪ extra/unallowed output).
  Score 3: Partial - Covers some core elements,
      ↪ but the main requested functionality
      ↪ or multiple required details are
      ↪ missing. Noticeable format/constraint
      ↪ violations.
  Score 4: Mostly compliant - Implements the
      ↪ task and required interfaces. Minor
      ↪ deviations only (e.g., small extra
      ↪ text, slight format drift, optional
      ↪ features omitted).
  Score 5: Fully compliant - Directly addresses
      ↪ the prompt; satisfies all requirements
      ↪ and constraints (language,
```

```
      ↪ APIs/signatures, I/O format, code-only
      ↪ vs. code+text). No extraneous or
      ↪ prohibited content.
  Notes for the grader-model: Judge adherence
      ↪ to instructions separately from
      ↪ correctness/style. If the prompt says "
      ↪ code only," any extra prose lowers the
      ↪ score. When in doubt between two
      ↪ scores, choose the lower one.

Humble: |-
  [Does the response avoid
      ↪ self-praise/overconfidence and show
      ↪ appropriate humility?]
  Score 1: Boastful/overconfident - Self-praise
      ↪ or superlatives ("state-of-the-art", "
      ↪ best ever"). Absolute claims ("100%
      ↪ guaranteed") without caveats.
      ↪ Dismissive of alternatives.
  Score 2: Arrogant tone - Frequent
      ↪ overconfident wording ("obviously", "
      ↪ trivial"), asserts correctness without
      ↪ context. Rare or token caveats.
  Score 3: Neutral/plain - No boastfulness, but
      ↪ also no acknowledgment of limits when
      ↪ relevant. Flat tone; neither
      ↪ self-praise nor self-critique.
  Score 4: Humble-accurate - Respectful tone,
      ↪ avoids superlatives. Uses measured
      ↪ language ("likely", "based on"). Notes
      ↪ assumptions/uncertainties or
      ↪ improvement points when relevant.
  Score 5: Constructively humble - All of 4,
      ↪ plus proactive, concise caveats
      ↪ (scope, trade-offs), gives
      ↪ credit/alternatives, and frames
      ↪ suggestions without self-promotion.
  Notes for the grader-model: Judge tone only,
      ↪ not technical correctness. Do not
      ↪ penalize clear, confident statements
      ↪ that are appropriately scoped and
      ↪ non-boastful.
```

E Mean, Median, Min, Max, and Invalid Data reporting for Comment Evaluation

Table 11: LLM-as-judge statistics for Go. Nominal N=500 per model; Invalid = $\max(0, 500 - \text{count})$.

Model	Mean	Median	Min	Max	Valid	Invalid
Qwen/Qwen2.5-14B-Instruct	4.569	5.0	1	7	499	1
WizardLMTeam/WizardCoder-15B-V1.0	6.114	7.0	1	7	220	280
bigcode/starcoder2-15b	6.417	7.0	1	7	338	162
codellama/CodeLlama-13b-Instruct-hf	6.427	7.0	1	7	487	13
deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct	6.225	7.0	1	7	497	3
microsoft/Phi-3-medium-128k-instruct	6.665	7.0	1	7	167	333

Table 12: LLM-as-judge statistics for Python. Nominal N=500 per model; Invalid = $\max(0, 500 - \text{count})$.

Model	Mean	Median	Min	Max	Valid	Invalid
Qwen/Qwen2.5-14B-Instruct	5.185	5.0	1	7	498	2
WizardLMTeam/WizardCoder-15B-V1.0	5.746	7.0	1	7	256	244
bigcode/starcoder2-15b	6.032	7.0	1	7	311	189
codellama/CodeLlama-13b-Instruct-hf	6.627	7.0	1	7	482	18
deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct	6.377	7.0	1	7	486	14
microsoft/Phi-3-medium-128k-instruct	6.765	7.0	3	7	341	159

Table 13: LLM-as-judge statistics for Rust. Nominal N=500 per model; Invalid = $\max(0, 500 - \text{count})$.

Model	Mean	Median	Min	Max	Valid	Invalid
Qwen/Qwen2.5-14B-Instruct	5.004	5.0	1	7	500	0
WizardLMTeam/WizardCoder-15B-V1.0	5.934	7.0	1	7	121	379
bigcode/starcoder2-15b	6.346	7.0	1	7	381	119
codellama/CodeLlama-13b-Instruct-hf	6.494	7.0	1	7	476	24
deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct	5.967	7.0	3	7	487	13
microsoft/Phi-3-medium-128k-instruct	6.792	7.0	5	7	178	322

Table 14: LLM-as-judge statistics for the "OOD-DSL" domain. Nominal N=525 per model; Invalid = $\max(0, 525 - \text{count})$. Rows with Valid$>525$ are kept as-is and Invalid set to 0.

Model	Mean	Median	Min	Max	Valid	Invalid
Qwen/Qwen2.5-14B-Instruct	5.375	6.0	1	6	525	0
WizardLMTeam/WizardCoder-15B-V1.0	7.000	7.0	7	7	17	483
bigcode/starcoder2-15b	6.709	7.0	1	7	385	115
codellama/CodeLlama-13b-Instruct-hf	6.973	7.0	1	7	515	0
deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct	6.555	7.0	4	7	503	0
microsoft/Phi-3-medium-128k-instruct	6.437	6.0	6	7	71	429