

Sketch-Guided Multi-Objective Alignment for Code Language Models: Optimizing Correctness, Efficiency, and Algorithmic Reasoning

Siqi Qiao

East China University of Science and Technology
Shanghai, China
23010342@mail.ecust.edu.cn

Qingchun Hu

East China University of Science and Technology
Shanghai, China
hqingchun@ecust.edu.cn

Fangli Ying*

East China University of Science and Technology
Shanghai, China
yfangli@ecust.edu.cn

Tingxi Li

East China University of Science and Technology
Shanghai, China
23013074@mail.ecust.edu.cn

Abstract

Code language models (CLMs) have achieved remarkable progress in program synthesis, with recent advances focusing on two major objectives: functional correctness and runtime efficiency. However, existing methods evaluate code solely by its runtime behavior, not by the soundness of the underlying problem-solving strategy. To address this gap, we propose a lightweight multi-objective alignment framework that integrates sketch-guided chain-of-thought (S-CoT) reasoning to strengthen intermediate algorithmic thinking. We introduce a minimal sketch generation step that prompts the model to produce a high-level algorithmic strategy before writing code, and quantify reasoning quality into a learnable preference signal. We then jointly optimize for correctness, efficiency, and reasoning quality within a unified direct preference optimization (DPO) framework. Extensive experiments on two commonly used datasets demonstrate that our approach can effectively mitigate logical inconsistencies and improve generalization to complex problems.

CCS Concepts

• Computing methodologies → Artificial intelligence.

Keywords

Algorithmic Direct Preference Optimization, Algorithm quality scoring, Multi-objective optimization, Supervised Fine-Tuning, Direct Preference Optimization, Model alignment, Post training

1 Introduction

Code language models (CLMs) have made remarkable strides in automatic program synthesis, driven by advances in large language models and dedicated prompting and alignment techniques [5]. Powered by large-scale pre-training on extensive code repositories, existing CLMs, including Codex [2], CodeLlama [9], and StarCoder [7], have demonstrated strong performance on code generation. To improve practical usability, recent research has increasingly

focused on model alignment toward two critical objectives: *functional correctness* (whether code passes all unit tests) and *runtime efficiency* (execution speed in real deployment). A particularly promising line of work leverages self-generated preference data to align code models without relying on expensive human annotations or proprietary large models. For instance, Code-Optimise [4] proposes a lightweight framework that samples diverse solutions, automatically annotates correctness and runtime through execution, and fine-tunes models via Direct Preference Optimization (DPO). Similarly, CodeDPO [11] introduces a self-generation-and-validation mechanism inspired by PageRank to score and rank code snippets and test cases mutually, further enhancing alignment quality by jointly optimizing correctness and efficiency. These execution-driven methods have significantly advanced the state of the art by grounding model alignment in objective and automated behavioral signals.

Despite these advances, a critical limitation persists in existing alignment pipelines: *execution-based signals only reflect code output behavior, not the quality of the underlying algorithmic reasoning*. For example, Thinking-in-Program (TIP) [10] show that guiding large language models to generate explicit algorithmic sketches, which are high-level solution blueprints built with programming structures, substantially improves the correctness, readability, and maintainability of generated code. These studies confirm that algorithmic reasoning quality is an independent and critical indicator of high-quality code that directly impacts generalization and robustness. However, no existing alignment framework integrates algorithmic reasoning quality as a learnable preference signal alongside correctness and efficiency.

This limitation becomes even more evident given recent progress in chain-of-thought reasoning for code [6]. As highlighted by this structured reasoning research for code generation, high-quality code synthesis requires not only valid syntax and correct execution but also sound *programmatic thinking* [3], which can be used to design logical, generalizable solution strategies with sequence, branch and loop structures. A code snippet may pass all given test cases but still rely on brute-force logic, redundant control flows, or hard-coded edge cases, resulting in fragile and non-scalable implementations. Existing alignment frameworks treat code as a black box and overlook this fundamental dimension of code quality.

*Corresponding author.

However, existing CoT methods force the model to generate verbose reasoning for every problem, regardless of its difficulty. This leads to the well-documented "overthinking" phenomenon [12], where models introduce errors into reasoning chains that would not have occurred if they had generated code directly. It also increases inference latency and computational cost, making it impractical for production use. No existing work has successfully integrated algorithmic reasoning quality into execution-centric alignment while avoiding these pitfalls.

To fill this gap, we propose a *sketch-guided multi-objective alignment framework* that unifies three optimization targets: functional correctness, runtime efficiency, and algorithmic reasoning quality. We observe that algorithmic reasoning can be made explicit and measurable through a lightweight sketch generation step. Instead of forcing the model to generate a full natural language explanation, we ask it to produce a 1-2 sentence high-level sketch of its algorithmic strategy before writing code. This sketch is concise enough to avoid overthinking, yet structured enough to capture the core of the model's reasoning.

Building on this insight, we present a sketch-guided multi-objective alignment framework that extends direct preference optimization (DPO) with a third orthogonal signal: algorithmic reasoning soundness. Our framework retains all the advantages of execution-centric methods: First, we employ execution-based preference learning framework from Code-Optimise [4] and CodeDPO [11]. It uses self-generated preference data, requires no human annotation, and is computationally lightweight. Then, we build a novel structured algorithmic sketching approach to guide the preference learning. It adds only a single sketch generation step and an LLM-as-Judge automatic scoring module that evaluates the quality of the sketch-code pair. Finally, we adopt the dynamic solution selection for stable and scalable DPO alignment for code language models. To the best of our knowledge, this is the first work to integrate explicit algorithmic reasoning into a self-generated preference alignment framework for CLMs. Our framework retains all the scalability and simplicity of execution-centric alignment, while producing code that is not only correct and fast, but also robust and generalizable. Our core contributions are summarized as follows:

- We identify algorithmic reasoning soundness as a critical missing dimension in CLM alignment, and we design a formal *algorithmic quality scoring rubric* that quantifies solution excellence beyond execution results. It makes algorithmic reasoning explicit and measurable, avoiding the overthinking and performance degradation of traditional CoT methods.
- We further develop an LLM-as-Judge automatic scoring module to provide fine-grained, interpretable evaluations of algorithmic quality for structured sketches, yielding informative supervision that explains *why* one solution is superior to another.
- We propose a unified alignment pipeline with a novel *Good-versus-Bad (GvB)* preference paradigm that jointly optimizes algorithmic quality, correctness, and efficiency in a single DPO framework. The initial experimental results on the MBPP benchmark demonstrate that our approach outperforms the baseline alignment methods across all three

objectives. By elevating algorithmic reasoning to a core optimization target, our model achieves stronger generalization to complex and unseen problems, verifying that optimizing *how models reason about code* leads to more robust, reliable, and high-quality program synthesis.

2 Methodology

Inspired by Code-Optimise [4], our framework inherits its DPO-based learning paradigm's core strengths, with targeted extensions to incorporate algorithmic reasoning. To address the critical limitation of existing execution-centric alignment frameworks, where they often fail to account for algorithmic reasoning quality, we firstly propose a novel sketch-guided sampling that makes algorithmic reasoning explicit and measurable. For each problem, we sample 100 candidate solutions, each consisting of a high-level algorithmic sketch followed by executable code, instead of the direct code sampling. This approach addresses the critical gap of overlooked algorithmic reasoning quality in existing alignment frameworks. Then, we develop an LLM-as-Judge module that evaluates the sketch-code pair (from 0 to 10 rating). Finally, we integrate a Good-versus-Bad (GvB) preference signal into Multi-Objective Preference Optimization, enabling joint optimization of correctness, efficiency, and reasoning quality. Additionally, we adopt the Dynamic Solution Selection (DSS) to mitigate overfitting. The overall pipeline of our proposed framework is illustrated in Figure 1, which visually delineates the integration of these novel components with the baseline DPO workflow.

2.1 Sketch-Guided Solution Sampling

For each problem x in the seed dataset $D_{\text{seed}} = x_i, ut_{i=1}^N$, where ut_i denotes the unit tests for problem x_i . we first prompt the CLM [7] to generate 100 candidate solutions, instead of direct code generation from CLM, we use the following sketch-guided prompt template shown in Figure 2.

This prompting strategy prompts the model to explicitly reason out its high-level algorithmic logic before producing concrete code. Each generated solution is structured as a pair $y = (s, c)$, where s represents the high-level algorithmic sketch and c denotes the executable code implementation. Such a design stabilizes the model's reasoning process by forcing it to lock down a general solution strategy prior to coding details. Moreover, it creates a structured evaluation target for measuring algorithmic quality, which is far more credible than directly assessing unstructured raw code especially for problems requiring complex logical reasoning.

After solution sampling, we follow the data filtering rule proposed in prior work and discard problems with fewer than two passing solutions or fewer than one failing solution. This preprocessing step guarantees valid preference pair construction for all three optimization objectives.

2.2 Algorithmic Score Annotation

To evaluate intrinsic algorithmic quality beyond basic functional correctness, we introduce an LLM-as-Judge module to quantify reasoning competence with an algorithmic score ranging from 0 to 10. In the annotation stage, every candidate solution is sent to an LLM evaluator. We leverage the GLM-4-Air model via standard

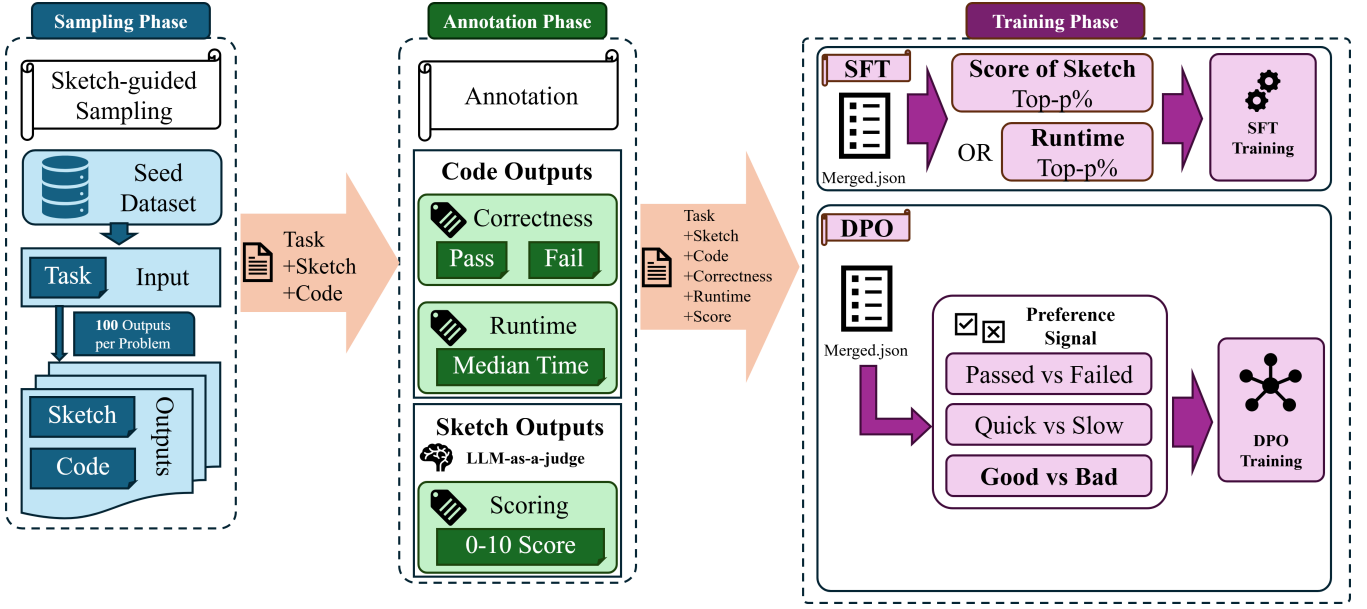


Figure 1: Overview of our sketch-guided multi-objective alignment framework. **Sampling Phase:** For each problem in the seed dataset, we prompt the CLM to generate 100 candidate solutions, each consisting of a high-level algorithmic sketch followed by executable code. **Annotation Phase:** Each solution is evaluated along three dimensions: functional correctness (pass/fail) via unit test execution, runtime performance (median execution time), and algorithmic quality (algorithmic score) assigned by an LLM-as-Judge evaluator. **Training Phase:** The annotated solutions are used in two training paradigms: (1) SFT, which selects top- k % solutions ranked by algorithmic score or runtime as training targets; (2) DPO, which constructs preference pairs from three orthogonal signals—Passed vs. Failed (PvF), Quick vs. Slow (QvS), and Good vs. Bad (GvB)—to jointly optimize correctness, efficiency, and algorithmic reasoning quality.

LLM API to comprehensively assess each solution along three key dimensions: time and space complexity optimality, code readability and clarity, and edge case coverage capability. The detailed prompt and scoring rubric, where 0 indicates no meaningful code and 10 denotes perfect optimality, are provided in supplementary file for reproducibility. We further assess the evaluator against blinded expert ratings and repeated LLM evaluations in Section 3.3.

As a result, for each problem x in our self-generated dataset D_{train} , we obtain a set of candidate solutions $\mathcal{Y}(x)$, each annotated with (1) functional correctness, which is pass or fail, (2) runtime performance, characterized by mean and standard deviation, and (3) algorithmic score, which is only assigned to solutions that pass all unit tests.

2.3 Good-versus-Bad (GvB) Preference Signal

We define the **Good versus Bad (GvB)** signal to distinguish solutions that exhibit sound algorithmic thinking from those that, while functionally correct, rely on brute-force or poorly generalisable approaches. Formally, for a problem x , we filter the candidate solution set $\mathcal{Y}(x)$ to keep only solutions that both pass all unit tests and have a valid algorithmic score (i.e., $\text{score} \neq -1$). We then divide this filtered subset into two mutually exclusive groups using a threshold τ :

$$\mathcal{G}(x) = \{y \in \mathcal{Y}(x) \mid \text{score}(y) \geq \tau\}, \quad (1)$$

$$\mathcal{B}(x) = \{y \in \mathcal{Y}(x) \mid \text{score}(y) < \tau\}. \quad (2)$$

When $\mathcal{G}(x)$ or $\mathcal{B}(x)$ is empty, the problem cannot provide a valid GvB preference pair and is discarded from the corresponding training split. Throughout our experiments, we set $\tau = 6$, which empirically separates “acceptable but clearly suboptimal” solutions ($\text{score} < 6$) from those with sound algorithmic choices (≥ 6).

We retain both the QvS and PvF criteria from prior work [4], and introduce GvB as a third orthogonal signal that addresses their critical limitation: inability to evaluate algorithmic reasoning quality. GvB distinguishes between functionally correct and efficient solutions that differ in underlying algorithmic soundness, enabling the model to avoid suboptimal implementations and generate more generalizable code for complex logical problems.

2.4 GvB in Supervised Fine-Tuning

To align the model with algorithmic quality objectives during supervised fine-tuning (SFT), we curate high-quality training targets by ranking solutions based on their algorithmic scores. Given a retention ratio $p\%$ (we experiment with $p = 25$ and $p = 100$), we first discard solutions whose `algorithm_score` is None or -1 , then sort the remaining passed solutions in descending order of their scores (breaking ties by runtime votes). We retain the top $p\%$ of

```

1  '''You are an expert Python programmer. For each task, first explain your
   algorithmic approach in 2-4 sentences of plain English, then provide
   the complete Python implementation in a single fenced code block.

2
3  ### Task:
4  def add(a, b):
5      """Write a function that returns the sum of two numbers."""
6
7  ### Approach:
8  The problem is straightforward: take two inputs and return their sum using
   the built-in + operator. No edge-case handling is needed because
   Python natively handles ints, floats, and mixed types.
9
10 ### Solution:
11 ```python
12 def add(a, b):
13     return a + b
14 ```
15
16 ### Task:
17 def is_palindrome(s):
18     """Write a function to check whether the given string is a palindrome
   ."""
19
20 ### Approach:
21 A palindrome reads the same forwards and backwards. The simplest correct
   algorithm compares the string with its reverse in O(n) time and O(n)
   space. We could also use two pointers for O(1) extra space, but
   slicing is more idiomatic for short strings.
22
23 ### Solution:
24 ```python
25 def is_palindrome(s):
26     return s == s[::-1]
27 ```
28
29 ### Task:
30 {problem}
31
32 ### Approach:
33 '''

```

Figure 2: Prompt template of sketch-guided solution sampling

solutions. Training proceeds under two modes controlled by the augment flag:

Static selection: A single solution is uniformly sampled from the retained set and fixed as the sole positive training target for all epochs.

Dynamic solution selection (DSS): All solutions in the top $p\%$ are kept as a candidate pool. A new target is randomly sampled per problem at the start of each epoch, implementing the dynamic augmentation strategy from [4].

In both modes, our SFTCollator receives the tokenised candidate pool and randomly picks one completion per forward pass. The standard SFT loss is applied:

$$\mathcal{L}_{\text{SFT}}(\pi_\theta) = -\mathbb{E}_{(x,y) \sim D} [\log \pi_\theta(y | x)], \quad (3)$$

where (x, y) is a prompt–solution pair formatted as “### Question: $\{x\}$ \n ### Answer: $\{y\}$ ”. By training only on the highest-scoring (algorithmically superior) solutions, we steer the model toward producing code that embodies better algorithmic design.

2.5 GvB in Direct Preference Optimisation

For DPO, GvB enables principled construction of preference pairs independent of runtime measurements. For each problem x that has at least one solution in $\mathcal{G}(x)$ and one in $\mathcal{B}(x)$, we randomly sample a chosen response $y_w \in \mathcal{G}$ and a rejected response $y_l \in \mathcal{B}(x)$ to

form the preference pair (x, y_w, y_l) . We apply DSS here as well, resampling pairs at each epoch to enhance generalization.

A key advantage of GvB-based pairs is that they are exclusively derived from functionally correct solutions. This forces the model to discriminate between nuanced levels of algorithmic quality rather than merely distinguishing correct from incorrect outputs. Unlike QvS and PvF, which only capture observable execution outcomes, GvB reveals intrinsic differences in reasoning soundness. It thus complements the runtime-driven and correctness-driven signals, enabling multi-objective optimization that elevates all three dimensions of code quality.

3 Experiments

In this section, we evaluate our DPO-based approach augmented with the proposed GvB algorithmic preference signal, on the standard MBPP code generation benchmark. Our experiments aim to answer four research questions: (i) Does GvB consistently improve functional correctness over both the base model and standard execution-centric fine-tuning? (ii) Can GvB reduce execution runtime and improve algorithmic quality? (iii) How does GvB interact with different training paradigms (SFT vs. DPO)? (iv) How reliably does the LLM evaluator reproduce expert algorithmic-quality judgments and the resulting GvB labels?

3.1 Implementation Details and Setup

Base Model and Sampling. For direct and fair comparison with the baseline [4], we adopt the identical StarCoderBase-1B [7] as our backbone CLM. For each problem in the MBPP training and validation splits, we generate 100 candidate solutions with backbone CLM using multinomial sampling with temperature $t = 0.6$, and maximum length 512 tokens. This temperature balances functional correctness and solution diversity, ensuring we obtain a wide range of algorithmic approaches for each problem. The generation is performed with DeepSpeed inference[8] [1] using the full model in FP16.

Annotation. Each generated solution is executed against the corresponding unit tests to determine functional correctness (pass/fail). For passing solutions, we measure runtime (averaged over 50 stable runs, following the same measurement protocol as [4]) and obtain an *algorithmic score* (0–10) from the GLM-4-Air LLM-as-Judge evaluator. Failed solutions are automatically assigned a score of -1 and excluded from GvB preference pair construction.

Evaluator Validation Protocol. To assess the reliability of this automatic supervision, we randomly sample 200 sketch-code solutions from 20 problems, with 10 solutions per problem. Two human experts independently assign a 0–10 algorithmic-quality score under a blinded protocol without access to the LLM outputs. We additionally evaluate each solution five times with the same LLM scoring rubric. Human agreement is measured between the two expert scores, while LLM–human agreement is computed against their mean score. We also report agreement after converting scores into the downstream GvB labels using the same threshold $\tau = 6$.

Data Filtering. Following [4], we retain only problems with at least two passing solutions and one failing solution to enable valid preference pair construction for all three objectives. The resulting dataset statistics are summarized in Table 1.

Fine-Tuning Configurations. We train all models under both SFT and DPO paradigms with identical hyperparameters to ensure fair comparison: **SFT**: Supervised fine-tuning on the top $p\%$ of solutions selected either by runtime (SPD) or by algorithmic score (ALG). We test $p = 25$ (SFT-25) and $p = 100$ (SFT-100). We further enable dynamic solution selection (DSS)[4] by randomly drawing a training target from the top- $p\%$ pool each epoch. **DPO**: Direct preference optimisation with the following preference criteria:

- **QvS** (Quick vs. Slow): a randomly chosen passed solution vs. a faster passed solution.
- **PvF** (Passed vs. Failed): a passed solution vs. a failed solution.
- **ALL**: combination of QvS and PvF pairs.
- **GvB** (Good vs. Bad algorithmic score): a passed solution with `algorithm_score` ≥ 6 vs. one with `score` < 6 .

DSS is applied to all DPO variants, with pairs resampled each epoch. All models are trained for 30 epochs with a learning rate of 5×10^{-7} , linear warmup (10%) and cosine decay, batch size 2 per GPU, gradient accumulation 16 steps, and DeepSpeed ZeRO-3 offloading. We select the checkpoint with the lowest validation loss.

Evaluation Metrics. We evaluate three core dimensions of code quality aligned with our optimization objectives: (1) We assess *functional correctness* via `pass@k` ($k \in \{1, 10, 100\}$) estimated from 100 sampled solutions per problem using the unbiased estimator. (2) *Runtime* is measured as the median execution time (in microseconds) of all correct solutions. (3) *Algorithmic quality* is calculated as the average algorithmic score assigned by the teacher model to all passing solutions.

3.2 Self-Generated Dataset Statistics

Table 1 summarizes the dataset statistics before and after filtering. While nearly half of the problems are discarded due to the strict filtering criteria, the remaining corpus provides thousands of diverse training examples sufficient for robust model alignment.

3.3 Reliability of the LLM-as-a-Judge Evaluator

Table 2 reports the evaluator validation results, addressing RQ4. The two experts show strong agreement (Spearman’s $\rho = 0.894$), and the LLM evaluator also demonstrates strong overall correlation with the mean expert score ($\rho = 0.869$), with a mean absolute error of 1.166 points on the 0–10 scale. After thresholding the scores at $\tau = 6$, the LLM-derived and expert-derived GvB labels agree on 79.5% of the cases, corresponding to moderate chance-corrected agreement (Cohen’s $\kappa = 0.504$) and a macro-F1 of 0.747. Repeated evaluation yields a mean within-solution standard deviation of 0.792 and a GvB label flip rate of 35.0%, indicating non-negligible variability for solutions near the decision threshold. Overall, the evaluator is a scalable but noisy supervision source: it aligns well with expert scores in aggregate and achieves moderate agreement on the downstream GvB labels, but should not be interpreted as a deterministic substitute for expert judgment.

Table 1: Statistics of self-generated preference data (StarCoderBase-1B). The slight discrepancies between the sample data and the data in the Code-Optimise paper stem from different sampling seeds and additional filtering during the algorithm scoring phase.

Split	Total	Filtered	Ratio (%)
<i>Problems</i>			
train(Ours)	384	177	46.09
train(CodeOpts)	384	183	47.66
validation(Ours)	90	43	47.78
validation(CodeOpts)	90	40	44.44
<i>Solutions</i>			
train(Ours)	38400	14966	38.97
train(CodeOpts)	38400	15472	40.29
validation(Ours)	9000	3828	42.53
validation(CodeOpts)	9000	3533	39.26

Table 2: Reliability of the LLM-as-a-Judge evaluator on 200 sketch-code solutions. Human scores are averaged across two blinded experts.

Metric	Value
Human–human Spearman’s ρ	0.894
LLM–human Spearman’s ρ	0.869
LLM–human MAE	1.166
Mean repeated-score SD	0.792
GvB label agreement	79.5%
Cohen’s κ	0.504
GvB macro-F1	0.747
GvB label flip rate	35.0%

3.4 Experimental Results

Functional Correctness on MBPP. Figure 3 presents the `pass@k` scores on the MBPP test set, addressing RQ1. All SFT variants yield significant improvements in `pass@1` performance over the base model, regardless of whether solutions are ranked by runtime (SPD) or algorithmic score (ALG). Notably, algorithm-guided SFT (ALG) consistently outperforms runtime-guided SFT (SPD) across all `pass@k` ($k=1,10,100$), demonstrating that prioritizing algorithmic reasoning enhances functional correctness.

Execution Efficiency. Figure 4 compares the median runtime of all passing solutions on MBPP and HumanEval, addressing RQ2. DPO with GvB preferences reduces median runtime from 90.1227 microseconds (base model) to 86.8153 microseconds. This result confirms that optimizing for algorithmic quality indirectly improves execution efficiency: sound algorithmic design inherently yields faster code, even when runtime is not explicitly used as a preference signal.

Algorithmic Quality. Figure 5 reports average algorithmic scores across different optimization methods, addressing RQ2. Both SFT-ALG and DPO-GvB achieve significantly higher algorithmic scores

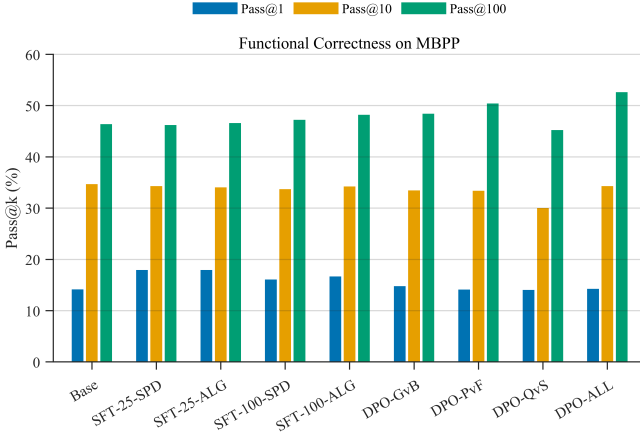
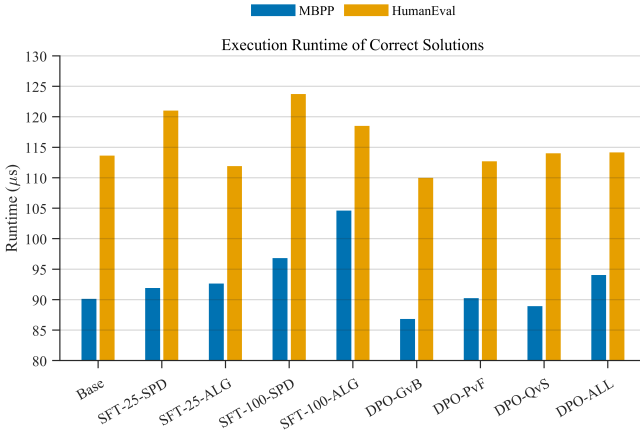


Figure 3: Pass@k(%) on MBPP test set


 Figure 4: Runtime (μ s) of correct solutions on the MBPP and HumanEval test sets.

than their execution-centric counterparts (SFT-SPD and DPO-QvS, respectively). Consistent improvements are observed on both MBPP and HumanEval datasets, verifying that GvB effectively enables the model to generate code with sound algorithmic reasoning, not just correct execution.

3.4.1 Impact of GvB over Base Selection. Figure 6 quantifies the marginal benefit of adding the GvB signal, addressing RQ3. Compared to the base model, all sketch-guided methods yield substantial improvements in Pass@1. Specifically, SFT-25-ALG (top 25% algorithmic solutions) achieves a 26.7% relative improvement in Pass@1, rising from 14.15% to 17.93%. This demonstrates that GvB complements both SFT and DPO paradigms, providing consistent gains across training strategies.

3.5 Conclusion and Future Work

This work addresses a critical limitation of existing execution-centric code alignment frameworks: their neglect of intrinsic algorithmic reasoning quality. We propose a lightweight sketch-guided multi-objective alignment framework that integrates algorithmic reasoning as a first-class optimization target alongside functional

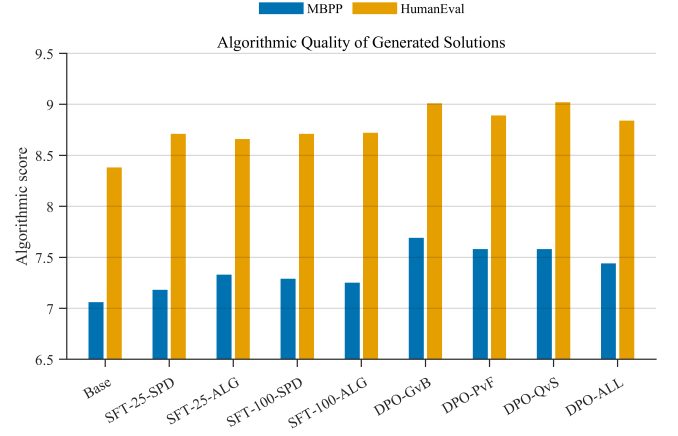


Figure 5: Algorithm performance of different models on the MBPP and HumanEval datasets

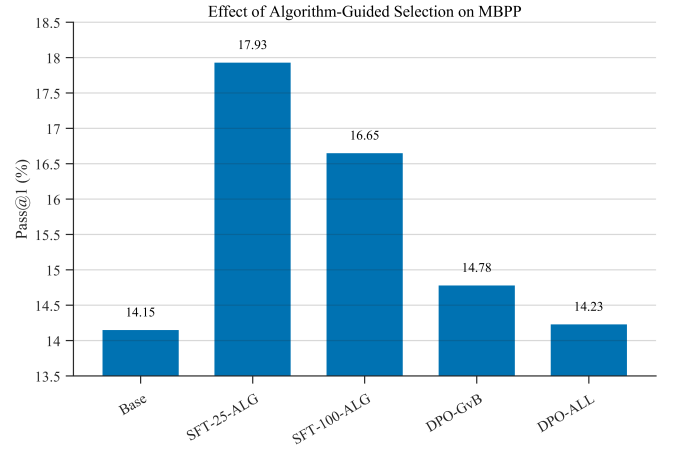


Figure 6: Pass@1 on MBPP with and without algorithmic selection. The GvB signal (ALG) lifts performance above the baseline.

correctness and runtime efficiency. Our approach combines sketch-guided sampling, an LLM-as-Judge evaluator with a formal scoring rubric, and the novel Good-versus-Bad (GvB) preference signal into a unified DPO pipeline. Extensive experiments on standard benchmarks demonstrate that our method consistently outperforms execution-centric baselines across all three objectives. A blinded human validation further shows strong aggregate correlation between the LLM evaluator and expert scores and moderate agreement on the downstream GvB labels, while repeated evaluations reveal that the automatic supervision remains noisy near the decision threshold.

In future work, we plan to extend this work to larger code models and diverse benchmarks, including cross-lingual generation and complex software engineering tasks. We will also improve the reliability of the LLM-as-Judge evaluator and explore integrating additional critical code quality dimensions such as security and maintainability into our multi-objective alignment framework.

References

- [1] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. 2022. DeepSpeed- Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. doi:10.1109/SC41404.2022.00051
- [2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [3] Anna Eckerdal, Michael Thuné, and Anders Berglund. 2005. What does it take to learn ‘programming thinking’?. In *Proceedings of the first international workshop on Computing education research*. 135–142.
- [4] Leonidas Gee, Milan Gritta, Gerasimos Lampouras, and Ignacio Iacobacci. 2025. Code-optimise: Self-generated preference data for correctness and efficiency. In *Findings of the Association for Computational Linguistics: NAACL 2025*. 79–94.
- [5] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology* 35, 2 (2026), 1–72.
- [6] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–23.
- [7] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
- [8] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (Virtual Event, CA, USA) (KDD ’20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. doi:10.1145/3394486.3406703
- [9] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
- [10] Jiaxin Wen, Jian Guan, Hongning Wang, Wei Wu, and Minlie Huang. 2024. Unlocking reasoning potential in large language models by scaling code-form planning. *arXiv preprint arXiv:2409.12452* (2024).
- [11] Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. 2025. Codedpo: Aligning code models with self generated and verified source code. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 15854–15871.
- [12] Yuqi Zhu, Ge Li, Xue Jiang, Jia Li, Hong Mei, Zhi Jin, and Yihong Dong. 2025. Uncertainty-guided chain-of-thought for code generation with llms. *arXiv preprint arXiv:2503.15341* (2025).

```

1 prompt = f"""You are an expert code reviewer. Evaluate the algorithmic quality of the following code solution for the given problem.
2
3 Problem description:
4 {problem_desc}
5
6 Code solution:
7 ...
8 {truncated_solution}
9 ...
10
11 Evaluate the code on the following dimensions:
12 - Correct algorithmic approach for the problem
13 - Time complexity (Big-O) optimality
14 - Space complexity optimality
15 - Code clarity and maintainability
16 - Proper handling of edge cases
17
18 You must use the following scoring guidelines to determine an integer score from 0 to 10:
19 0: No meaningful code
20 1: Only incomplete fragments; does not solve the problem
21 2: Barely addresses; almost entirely wrong
22 3: Major algorithmic mistakes; incorrect for many input categories
23 4: Partially correct; fails on certain inputs; notable logical flaws
24 5: Functional but brute-force where a better standard algorithm exists; misses important edge cases; messy code
25 6: Acceptable but clearly suboptimal; some edge cases mishandled
26 7: Good solution; correct but has moderate inefficiency or misses a few edge cases
27 8: Very good; correct and efficient, only minor issues (e.g. misses an obscure edge case)
28 9: Excellent; only trivial improvements possible (e.g. tiny constant-factor optimizations)
29 10: Perfect; optimal time and space complexity, flawless edge-case handling, exceptionally clear
30
31 First, provide a brief chain-of-thought analysis of the algorithm's strengths and weaknesses.
32 Then, output ONLY a single integer from 0 to 10. Do not include any other text, analysis, or explanation.
33
34 Format your response exactly as:
35 SCORE: <integer>"""

```

Figure 7: Prompt template

Supplementary Material: Prompt for Algorithmic Score Annotation (GvB)

To evaluate the intrinsic algorithmic quality of candidate solutions, we employ the GLM-4-Air model as an LLM-as-Judge evaluator, accessed via a standard OpenAI-compatible API. The model is tasked with assigning an integer score from 0 to 10 that quantifies the overall algorithmic merit of each solution. Evaluation is performed along five orthogonal dimensions: correctness of the underlying algorithmic approach, optimality of time complexity, optimality of space complexity, code clarity and maintainability, and robustness to edge cases. The evaluator is instructed to first generate a concise chain-of-thought analysis of the solution’s strengths and weaknesses, followed by a single integer score on the final line formatted exactly as SCORE: <integer>. To minimize token consumption, we truncate all code solutions to the first 2000 characters before submission to the evaluator. Solutions that fail to yield a parsable score after three retries with exponential backoff are assigned a score of -1 and excluded from subsequent GvB preference pair construction.

We present the full, unmodified prompt template employed across all experiments in this work in figure 7.

Algorithmic quality annotation is performed immediately after a solution is confirmed to pass all unit tests. We configure the LLM-as-Judge evaluator with *temperature* = 0 and *max_tokens* = 1024 to reduce sampling variability and encourage well-structured outputs.