

Reasoning Through Execution: Unifying Process and Outcome Rewards for Code Generation

Zhuohao Yu¹ Weizheng Gu¹ Yidong Wang¹ Xingru Jiang¹ Zhengran Zeng¹
Jindong Wang² Wei Ye¹ Shikun Zhang¹

Abstract

Large Language Models excel at code generation yet struggle with complex programming tasks that demand sophisticated reasoning. To bridge this gap, traditional process supervision relies on learned reward models requiring costly training data and suffering from reward misalignment, while outcome supervision fails for complex tasks needing coordinated intermediate steps. We introduce Outcome Refining Process Supervision, which unifies process and outcome supervision by leveraging executable verification: a tree-structured search framework generates strategic alternatives, profiles execution metrics, and scores candidates via self-critique mechanisms that integrate runtime feedback with reasoning. Experiments across 5 models and 3 benchmarks show consistent gains, with **26.9%** higher correctness and **42.2%** improved code efficiency. The results demonstrate that ORPS enables LLMs to overcome local optima in code generation, suggesting a promising direction for combining verifiable outcomes with structured reasoning to tackle complex challenges.

1. Introduction

Large Language Models (LLMs) have revolutionized code generation through their ability to synthesize programs from natural language specifications (Brown et al., 2020; Guo et al., 2024). However, complex programming tasks requiring multi-step algorithmic reasoning—such as implementing dynamic programming solutions or optimizing parallel computation patterns—remain challenging (Jiang et al., 2024b; Jimenez et al., 2023). These limitations persist even in state-of-the-art models (OpenAI, 2023; Touvron et al., 2023a), revealing a critical gap in current supervision paradigms.

¹Peking University, China ²William & Mary, Virginia, USA. Correspondence to: Wei Ye <wye@pku.edu.cn>.

Under review.

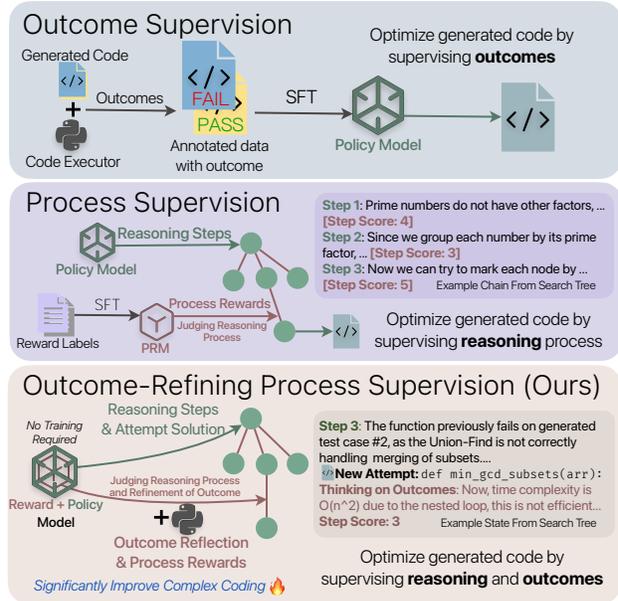


Figure 1: Comparison of outcome and process supervision.

As shown in Figure 1, traditional approaches follow two main paradigms: *outcome supervision*, which evaluates only final outcome quality (Chen et al., 2021b), and *process supervision*, which guides intermediate steps using learned Process Reward Models (PRMs) with search algorithms (Lightman et al., 2023). While PRMs have shown success in mathematical reasoning (Wang et al., 2024c; Chen et al., 2024a), their application to code generation faces fundamental challenges: (1) PRMs require expensive human annotations or distillations of other models on intermediate steps to train (Wang et al., 2024b); (2) Learned rewards suffer from *hallucination* (misjudging invalid steps as correct) (Huang et al., 2023; Stechly et al., 2024) and *reward hacking* (LLM exploiting superficial patterns to maximize scores) (Skalse et al., 2022); (3) No code-specific PRMs exist to date—existing implementations directly adapt math-focused PRMs might mismatch programming’s structured logic and data distribution.

Code generation presents a unique opportunity through *con-*

crete, verifiable signals. Unlike other domains where intermediate steps may be difficult to verify, code can be executed throughout development, providing objective feedback about both theoretical correctness and practical performance (Zhang et al., 2023; Shinn et al., 2024). Yet existing approaches using execution feedback (Shinn et al., 2024; Zhong et al., 2024) focus primarily on local improvements and debugging, missing opportunities for exploring fundamentally different algorithmic strategies.

We propose **Outcome-Refining Process Supervision**, a novel paradigm that treats the reasoning on refinement of outcomes itself as the process to be supervised. This approach differs fundamentally from existing self-improvement methods that focus on iterative refinement with execution feedback. Through a tree-structured exploration space, our framework maintains multiple reasoning trajectories simultaneously, enabling models to discover and refine diverse solution strategies. This structure allows models to explore different algorithmic approaches when initial attempts prove suboptimal (e.g., switching from brute-force to divide-and-conquer approaches), rather than being trapped in local optima (e.g., a brute-force solution that passes the test but is inefficient).

Our key insight is that execution feedback can serve as objective anchors for evaluating reasoning quality, eliminating the need for specially trained PRMs. This creates a natural synergy: execution outcomes (pass/fail) and performance metrics (runtime/memory) in conjunction with self-critique mechanisms ground the process rewards without human annotation. Tree-structured search maintains diverse solution trajectories, enabling recovery from early missteps through alternative approaches. Reasoning chains and code implementations refine each other—execution failures prompt theoretical re-analysis, while improved algorithms lead to better implementations.

To our knowledge, ORPS is the first process supervision framework specifically designed for code generation, requiring no PRM training while outperforming existing math-derived approaches. Experiments across 5 models and 3 datasets reveal:

- **Reasoning Over Model Scale:** Providing sufficient reasoning space is more crucial than model size for complex programming tasks - even smaller models like Qwen-7B achieve high success rates (80% Pass@1) when given room to explore and refine multiple solution strategies, outperforming its 14B counterpart
- **Learned PRM Elimination:** Combining execution feedback with self-critique mechanisms creates a more reliable verification system than learned reward models, without requiring expensive training data
- **Scalable improvement:** Our approach shows consis-

tent improvements in both success rates and solution efficiency compared to existing methods, particularly on complex tasks where current approaches struggle

Our key contributions include:

- We propose ORPS: a novel framework that unifies outcome and process supervision to tackle complex code problems.
- We found that eliminating specially trained PRMs with hybrid process rewards significantly enhances model’s reasoning ability to solve complex coding problems.
- ORPS achieves an average Pass@1 improvement of 26.9% across three datasets and five models, while reducing running time by 42.2% on average.

2. Related Work

2.1. Outcome Supervision vs Process Supervision

Outcome Supervision in language models traditionally evaluates and optimizes LLM outputs through three primary paradigms: (1) *Open-ended generation* tasks (e.g. instruction following) assessed via text similarity metrics with reference answers (Zhang et al., 2019) or LLM-as-a-judge scoring (Zheng et al., 2023; Wang et al., 2023c; Yu et al., 2024a); (2) *Constrained-output tasks* (multiple-choice, math problems) judged by exact answer matching (Brown et al., 2020; Hendrycks et al., 2020) of answer keys or solutions; and (3) *Code generation* where correctness depends on test case execution with generated programs (Chen et al., 2021b; Austin et al., 2021). These outcomes are then used to curate the data (e.g., selecting higher-quality conversations or code solutions) for further training of the model (Liu et al., 2023b). Such approaches share a critical limitation: they ignore the reasoning process that produced the output (Uesato et al., 2022). This proves particularly problematic for complex tasks where optimal solutions require coordinated intermediate steps (Lightman et al., 2023).

Process Supervision addresses this gap by optimizing *intermediate reasoning trajectories* through specially trained *Process Reward Models (PRMs)* that score each intermediate step (Uesato et al., 2022; Lightman et al., 2023). PRMs have proven particularly effective in domains requiring complex reasoning, such as math problems, where they guide search or sampling algorithms toward better reasoning trajectories and solutions (Wang et al., 2024c; Chen et al., 2024a; Wang et al., 2024b). While outcome supervision gives $R_{\text{outcome}}(y) = \mathbf{1}[\text{correct}]$, process supervision gives step-wise reward signals $R_{\text{process}} = \sum_{t=1}^T \text{PRM}(s_t | s_{1:t-1})$.

This approach have been predoominantly used in math reasoning tasks(Luo et al., 2024; Jiang et al., 2024a), but also

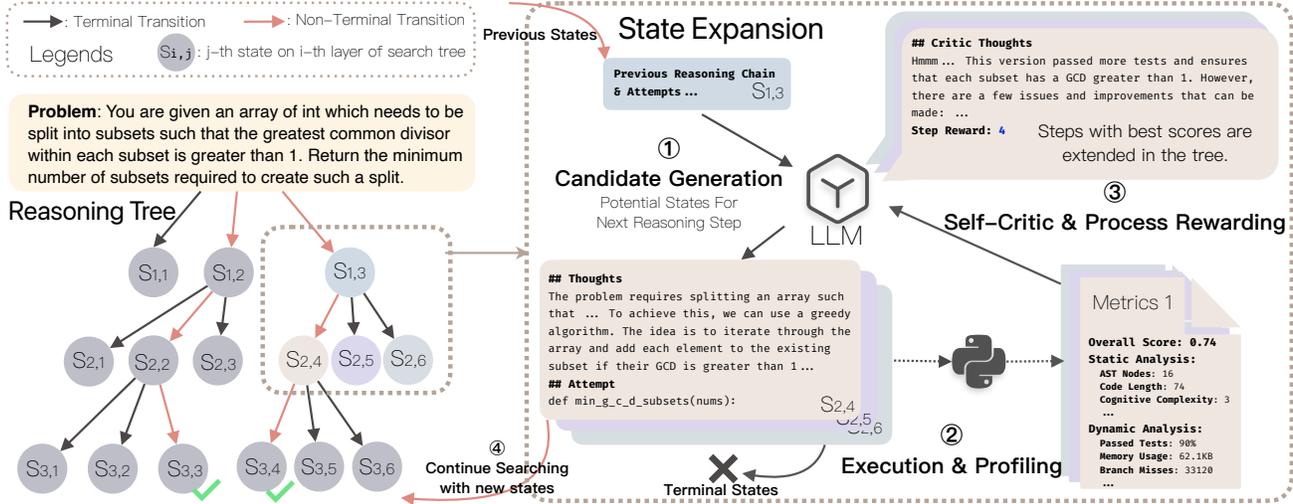


Figure 2: **Outcome-Refining Process Supervision** framework overview. A language model serves as both *programmer* and *critic* in a step-by-step reasoning process. Through beam search, the framework maintains multiple solution trajectories, where each state contains **reasoning chains**, **code implementations**, and **step reward**.

with limitations: 1) The requirement for dense human annotations to train reliable PRMs makes the approach expensive and time-consuming (Lightman et al., 2023). 2) The generalization capability of PRMs is often limited, as reasoning patterns can vary significantly across different tasks and domains. 3) When serving as judges, LLMs may produce unreliable evaluations due to hallucination (Hu et al., 2024; Li et al., 2024), particularly for complex tasks (Thakur et al., 2024). Recent studies show that LLMs cannot reliably self-correct (Huang et al., 2023) or self-validate without external verification (Stechly et al., 2024). These limitations motivate our approach of grounding process supervision in concrete, verifiable signals rather than learned judgments.

2.2. Execution-Driven Code Generation

Code generation is typically formulated as a sequence-to-sequence problem: given input specification x (including natural language description and test cases), generate a program y that correctly implements the required functionality (Jiang et al., 2024b). While most existing approaches treat this as a single-step generation process (Chen et al., 2021a), recent work has explored using execution feedback to guide code generation (Zhong et al., 2024; Zhang et al., 2023) or use CoT prompting to improve correctness (Shinn et al., 2024).

Although these execution-guided approaches show promise, our experiments indicate they are insufficient for complex programming tasks that require deeper reasoning. While execution feedback is easy to measure, it alone provides little guidance on how to improve solutions that fail or how to make working solutions more efficient. More importantly,

it offers no feedback during the intermediate stages of development, when early course corrections could prevent cascading errors.

Consider implementing an efficient sorting algorithm: a model might write code that passes all test cases but uses an inefficient $O(n^2)$ approach. Outcome supervision would mark this as a success, missing the opportunity to guide the model toward a more optimal $O(n \log n)$ solution. Similarly, if the code fails, a sparse “fail” signal provides no insight into whether the error lies in the algorithmic approach, the implementation details, or edge case handling. These limitations of both process and outcome supervision highlight the need to rethink how to supervise the development of complex programs, where both theoretical understanding and practical implementation must evolve together.

3. Methodology

We propose a framework that unifies process supervision with outcome supervision by combining *reasoning*, *code implementation*, and *execution verification* into a single tree-structured search process.

When tackling coding tasks, particularly complex ones, it is challenging for a model to generate a fully correct solution on the first attempt. Instead, LLMs often produce imperfect yet heuristically valuable code, requiring iterative self-correction to eventually arrive at a correct implementation. Each iteration of code refinement can be considered a step in the problem-solving process.

As formalized in Algorithm 1 and illustrated in Figure 2, at step t , the node in a given search beam represents a

Algorithm 1 Outcome-Refining Process Supervision

```

Input: Problem  $x$ , Unit Tests  $U$ , Model  $\mathcal{M}$ , Beam size  $K$ , Steps  $T$ ,
Candidates  $N$ , Process Reward Weight  $\alpha$ , Outcome Reward Weight  $\beta$ 
Initialize  $\text{beam}_0 \leftarrow \{(x, \emptyset)\}$  {Start with problem description}
for step  $t = 1$  to  $T$  do
  paths  $\leftarrow \emptyset$  {Initialize reasoning paths}
  for state  $s$  in  $\text{beam}_{t-1}$  do
    chain  $\leftarrow s.\text{reasoning\_chain}$  {Copy current reasoning chain}
    candidates  $\leftarrow \mathcal{M}_{\text{reason}}(x, \text{chain}, N)$  {Generate  $N$  pairs}
    for candidate in candidates do
      reasoning, code  $\leftarrow$  Extract from candidate
      newchain  $\leftarrow$  chain  $\oplus$  reasoning  $\oplus$  code
      feedback, outcome_rew  $\leftarrow$  execute_and_profile(code,  $U$ )
      newchain  $\leftarrow$  newchain  $\oplus$  feedback  $\oplus$  outcome_rew
      crit, process_rew  $\leftarrow \mathcal{M}_{\text{critic}}(\text{newchain}, \text{code}, \text{feedback})$ 
      newchain  $\leftarrow$  newchain  $\oplus$  crit  $\oplus$  process_rew
      step_score  $\leftarrow \alpha \times \text{process\_rew} + \beta \times \text{outcome\_rew}$ 
      paths  $\leftarrow$  paths  $\cup \{(\text{newchain}, \text{step\_score})\}$ 
    end for
  end for
   $\text{beam}_t \leftarrow$  Select Top- $K$  paths with highest step_score
  if Any path in  $\text{beam}_t$  is complete then
    break
  end if
end for
Return: Best reasoning chain from  $\text{beam}_T$ 

```

state $s_t = (\mathcal{R}_t, C_t, F_t, \omega_t, K_t, \rho_t)$, where \mathcal{R}_t denotes the current reasoning chain, C_t the code implementation, F_t the execution feedback, ω_t the outcome reward score, K_t the self-critic reasoning and ρ_t the process reward score.

The search progresses through three phases :

1. **Candidate Generation.** The LLM is leveraged to do reasoning on how to refine or alternative strategies and then attempt to generate corresponding code implementations.
2. **Execution & Profiling.** Each candidate code is executed and profiled on unit tests, to measure key performance metrics, such as correctness, efficiency, and code quality.
3. **Self-Critic & Process Rewarding.** The LLM is prompted to generate a textual self-critic considering reasoning chain and execution metrics, then gives a numerical process reward score which is used to guide searching.

Unlike linear CoT approaches that commit to a single trajectory, this tree structure enables *parallel exploration* of divergent strategies—for instance, maintaining both greedy and dynamic programming approaches for optimization problems until empirical feedback identifies the superior solution.

3.1. Candidate Generation

The candidate generation phase expands the search tree by producing diverse refinements. For each node $s_{t-1} = (\mathcal{R}_{t-1}, C_{t-1}, F_{t-1}, \omega_{t-1}, K_{t-1}, \rho_{t-1})$, the LLM generates N successor candidates:

$$\{(r_t^{(j)}, c_t^{(j)})\}_{j=1}^N = \mathcal{M}(\mathcal{R}_{t-1}, C_{t-1}, F_{t-1}),$$

where $r_t^{(j)}$ represents incremental reasoning updates (e.g., ‘Adjust loop termination to prevent off-by-one errors’) or strategic pivots (e.g., ‘Replace recursion with iteration to avoid stack overflow’). This dual-generation mechanism ensures *algorithmic diversity*—maintaining competing approaches until empirical feedback resolves ambiguities.

3.2. Execution & Profiling Outcome Rewards

We execute the model-generated code $\{c_t^{(j)}\}_{j=1}^N$ on a set of predefined Unit tests U and execute them to obtain results. U may be either generated by the LLM itself based on the problem or provided by the dataset. We will discuss the results under different scenarios in subsection 4.1.

The outcome reward $\omega_t^{(j)}$ combines *dynamic analysis* (Ball, 1999) and *static analysis* (Nielson et al., 1999) metrics into a unified score, grounding solution quality in both runtime behavior and structural properties. We compute:

$$\omega_t^{(j)} = \sum_{k=1}^M \beta_k \cdot \text{normalize}(m_k^{(j)}),$$

We have correctness, execution time, CPU instruction count, page faults as dynamic analysis metrics and code length, AST node count, cyclomatic complexity and cognitive complexity as static analysis metrics. Due to space limitations, we put detailed introduction to each metric in Table 5.

This framework rewards solutions that balance *functional correctness* with comprehensive aspects—a brute-force implementation passing all tests would score highly in correctness but poorly in complexity metrics, incentivizing refinement toward optimal algorithms.

3.3. Self-Critic & Process Rewarding

The same model \mathcal{M} serves dual roles: generating reasoning along with solution candidates, and judging their viability. After executing $c_t^{(j)}$ to obtain $F_t^{(j)}$, the model produces a textual critique $k_t^{(j)}$ and a numerical process reward $\rho_t^{(j)}$:

$$(k_t^{(j)}, \rho_t^{(j)}) = \mathcal{M}(\mathcal{R}_{t-1} \oplus r_t^{(j)}, C_{t-1} \oplus c_t^{(j)}, F_t^{(j)}).$$

The process reward grounds *subjective evaluation* of the reasoning process by combining *objective execution metrics*. This hybrid scoring prevents reward hacking—a model cannot inflate rewards without corresponding improvements in verifiable execution outcomes. Candidates with high ρ but low ω signal outcome-process mismatches, which is a critical failure mode in conventional process supervision.

3.4. Unifying Process and Outcome Supervision

The beam search mechanism proceeds with Top-K successor states using a weighted *step score*: $q_t = \alpha\rho_t + \beta\omega_t$. Where $\alpha + \beta = 1$ governs the trade-off between theoretical soundness (ρ_t) and empirical effectiveness (ω_t). The framework balances functional correctness while preserving promising reasoning trajectories.

This formulation generalizes conventional supervision paradigms. When $\beta = 0$, the framework reduces to pure process supervision akin to mathematical reasoning approaches that prioritize stepwise correctness over answers during reasoning. Conversely, $\alpha = 0$ recovers outcome supervision’s focus on final code quality, similar to Best-of-N sampling but on a tree. Our unified perspective reveals these as endpoints on a continuum of supervision strategies, with the proposed α/β balance enabling simultaneous optimization of reasoning and outcome solution quality.

The synergy emerges through bidirectional feedback: execution outcomes ground reasoning process by identifying discrepancies between intended and actual behavior (e.g., test failures revealing flawed base cases in recursive algorithms), while process rewards guide exploration toward algorithmically superior implementations (e.g., recognizing that memoization could transform an $O(2^n)$ brute-force solution into an $O(n)$ dynamic programming approach).

4. Experiments

Our experimental evaluation aims to address three key questions: (1) How effective is our framework compared to existing approaches? (2) How does each component of our framework contribute to the overall performance? (3) What insights can we gain about the relationship between reasoning quality and code generation?

Table 1: **Dataset Statistics.** Characteristics of the programming benchmarks used in evaluation.

	LBPP <small>(Matton et al., 2024)</small>	HumanEval <small>(Chen et al., 2021b)</small>	MBPP <small>(Austin et al., 2021)</small>
# Test Problems	162	164	257 [†]
# Unit Tests	5.1	6.5	3.0
Solution Length [§]	627 / 3039	169 / 622	130 / 589
Contamination	New Dataset	18.9% [‡]	20.8% [‡]
Difficulty	Competitive Programming	Basic Functions	Basic Functions
Task Type	Algorithms	Func. Completion	Basic Prog.

[†]From sanitized version; [‡]Contamination results reported from Riddell et al. (2024); [§]Average/maximum characters in solution code.

4.1. Experimental Setup

Datasets. We evaluate on 3 programming benchmarks as shown in Table 1. LBPP is a recent complex programming dataset manually curated by human experts with competitive programming experience. HumanEval and MBPP are popular code generation benchmarks but could be trivial for current LLMs (Matton et al., 2024). Moreover, a significant proportion of the data is leaked in multiple pre-training corpora (Riddell et al., 2024). To ensure reproducibility, we report our detailed hyperparameters in Appendix A, we also open-source all our code and scripts.

Unit Tests. Our framework utilizes unit tests to verify and profile code solutions. We use LLM to generate these unit tests, which are then employed in computing the outcome reward $\omega_t^{(j)}$. However, self-generated unit tests do not always effectively assess code quality. Consequently, some prior works (Zhong et al., 2024) directly utilize dataset-provided unit tests. To facilitate a fair comparison with related approaches, we also consider using unit tests provided by datasets. In Table 2, methods using unit tests from datasets are denoted by (w/ T).

Baselines. We compare several strong baselines for code generation. For outcome supervision, Reflexion (Shinn et al., 2024) is a recent self-improvement approach that utilizes execution results to refine generated code. LDB (Zhong et al., 2024) extends this by incorporating debugger outputs, and intermediate variable values for iterative solution refinement. For test-time scaling, we implement Best-of-N sampling, which generates multiple solutions and selects the best one based on test outcomes. Since no existing process supervision methods have been designed specifically for code generation, we adapt a similar approach from mathematical reasoning (Luo et al., 2024) in comparison, which we include in our ablation studies.

4.2. Main Results

Table 2 shows the comparative results of our method and baselines, Figure 3 provides detailed multi-dimensional profiling of the performance of generated solutions with different methods.

Our results indicate significant improvements in both correctness and code quality metrics, especially on harder benchmarks. Even a smaller model (Qwen 7B), when paired with our method, could surpass its larger variant (Qwen 14B) without our method, suggesting that *providing sufficient reasoning space can be more effective than solely scaling model parameters* - which is significantly more computationally expensive. This finding has important implications for practical applications where computational resources are limited.

When compared to other execution-feedback and outcome

Table 2: **Main Results on Code Generation Benchmarks.** **Pass@1**: solutions passing all test cases. **Tests**: average test cases passed. **Valid**: solutions that compile and execute. **Time**: relative execution time, compared to the standard solution. Best results are in **bold** and second-best are underlined, every metric is in percentage.

Model/Method	LBPP (2024)				HumanEval (2021b)				MBPP (2021)			
	Pass@1↑	Tests↑	Valid↑	Time↓	Pass@1↑	Tests↑	Valid↑	Time↓	Pass@1↑	Tests↑	Valid↑	Time↓
Llama-3.1-8B-Instruct (2024)												
CoT	30.9	44.3	63.0	176.8	50.0	68.4	82.9	98.1	58.0	64.9	72.4	91.9
Reflexion	34.0	49.3	67.3	148.5	54.9	71.1	83.5	107.5	58.8	65.0	71.2	88.6
LDB (w/ T)	25.9	39.8	58.0	252.2	54.3	62.3	66.5	127.1	43.6	47.1	49.4	170.7
BoN	<u>46.9</u>	64.7	84.6	107.6	<u>71.3</u>	84.7	93.3	77.3	<u>73.5</u>	<u>79.9</u>	<u>86.4</u>	<u>72.1</u>
ORPS	45.9	<u>66.9</u>	<u>88.5</u>	<u>99.1</u>	70.3	<u>87.5</u>	<u>96.2</u>	<u>65.8</u>	71.8	78.2	84.3	84.5
ORPS (w/ T)	67.1	81.4	93.7	89.4	91.4	95.7	98.1	63.6	90.4	93.1	95.6	59.1
DeepSeek-Coder-7B-Instruct-v1.5 (2024)												
CoT	32.7	45.9	67.3	160.1	65.9	78.2	85.4	86.9	69.3	75.0	80.9	77.7
Reflexion	25.9	41.9	63.0	153.0	63.4	77.1	86.6	101.0	68.9	74.4	80.2	74.2
LDB (w/ T)	31.5	45.7	61.7	206.2	74.4	80.0	81.7	85.6	61.1	64.0	66.1	98.3
BoN	49.4	63.9	80.2	123.4	73.8	88.1	94.5	64.1	<u>74.3</u>	80.2	86.8	68.9
ORPS	<u>56.3</u>	71.1	88.0	89.4	<u>76.2</u>	<u>90.0</u>	<u>96.3</u>	<u>40.6</u>	73.2	80.3	87.5	46.8
ORPS (w/ T)	63.7	80.8	96.9	74.4	95.7	98.0	99.4	31.8	93.0	94.7	96.1	34.2
Qwen-2.5-Coder-7B-Instruct (2024)												
CoT	40.1	55.3	72.2	118.6	72.6	79.0	82.3	79.2	79.0	83.3	88.3	67.3
Reflexion	37.7	57.1	78.4	111.2	75.6	81.1	84.1	73.6	79.0	84.0	88.7	63.5
LDB (w/ T)	35.8	49.9	65.4	187.8	<u>87.8</u>	90.3	91.5	76.1	66.9	69.4	72.0	96.8
BoN	53.1	68.8	85.8	117.9	77.4	85.1	87.8	66.8	<u>82.9</u>	<u>87.2</u>	<u>91.8</u>	<u>62.6</u>
ORPS	59.9	<u>75.7</u>	<u>92.0</u>	84.1	79.9	<u>91.6</u>	<u>96.3</u>	<u>48.3</u>	76.7	82.4	88.3	68.0
ORPS (w/ T)	77.8	87.9	96.9	82.4	96.3	98.0	98.8	43.9	94.9	96.4	97.3	45.3
Qwen-2.5-Coder-14B-Instruct (2024)												
CoT	53.7	63.9	77.2	119.2	82.9	88.5	90.2	76.6	<u>84.0</u>	<u>87.4</u>	<u>91.1</u>	67.5
Reflexion	60.5	70.5	82.1	113.3	83.5	89.9	92.7	68.8	83.3	87.2	<u>91.1</u>	66.0
LDB (w/ T)	51.9	62.9	75.3	225.2	<u>89.6</u>	92.0	92.7	140.5	72.4	74.6	<u>76.3</u>	149.7
BoN	<u>61.7</u>	74.9	<u>90.7</u>	115.6	87.8	<u>93.9</u>	95.7	58.8	81.7	86.4	<u>91.1</u>	<u>58.4</u>
ORPS	<u>61.7</u>	<u>77.4</u>	<u>90.7</u>	84.8	81.7	91.3	<u>96.3</u>	41.5	76.3	82.0	87.9	58.8
ORPS (w/ T)	85.8	90.7	95.7	64.2	97.0	98.5	99.4	<u>43.8</u>	95.3	96.9	98.1	41.0
GPT-4o-Mini (2024)												
CoT	50.0	65.9	80.2	124.5	79.9	87.5	90.9	80.5	78.6	83.5	87.9	70.3
Reflexion	62.3	73.9	87.7	93.2	75.0	83.6	87.2	75.1	79.4	84.0	88.3	67.6
LDB (w/ T)	54.9	67.8	82.7	220.1	<u>88.4</u>	92.2	93.9	133.4	72.8	75.5	77.8	157.9
BoN	64.2	78.6	93.8	88.9	82.9	90.2	92.7	66.5	80.5	85.5	89.9	64.6
ORPS	<u>67.9</u>	<u>81.2</u>	<u>94.4</u>	<u>81.5</u>	84.8	<u>92.7</u>	<u>96.3</u>	<u>57.5</u>	80.2	<u>86.0</u>	<u>91.8</u>	<u>64.7</u>
ORPS (w/ T)	88.9	94.3	98.1	61.6	97.6	98.7	99.4	46.2	95.7	97.3	98.4	51.4

reward based methods like Reflexion and LDB, our approach consistently demonstrates superior performance regardless of test case access. This improvement stems from a fundamental difference in approach: while these outcome-based methods focus primarily on local information like resolving execution errors and reasoning in chain structure, our method provides LLMs with *broader reasoning space to reflect on higher-level aspects such as algorithm selection and problem properties by using process reward guided search*. For instance, LDB achieves 35.8% Pass@1 on LBPP with Qwen-7B with test case access, while our method reaches 77.8% under the same conditions.

Particularly noteworthy is the performance boost when models have access to gold unit tests from test datasets (without access to solutions). All models show drastic improvements on all metrics in this setting. For instance, Qwen-7B

achieves 77.8% Pass@1 on LBPP and 96.3% on HumanEval with test case access, compared to 59.9% and 79.9% without. This suggests that while our self-generated test cases may be relatively weak, *given feedback for higher quality test cases, models can effectively guide themselves through the reasoning process to generate significantly better code*.

Figure 3 further supports these findings through detailed profiling results, showing consistent improvements over baselines across all models in terms of code efficiency and quality metrics. Guided by our framework, models are capable of refining themselves to generate faster, more coherent code. However, we do observe a slight disadvantage on MBPP, particularly when comparing with Best-of-N sampling. This is less concerning given that, as shown in Table 1, MBPP consists of relatively simple problems with short solutions, and a significant portion (20.8%) of its test data

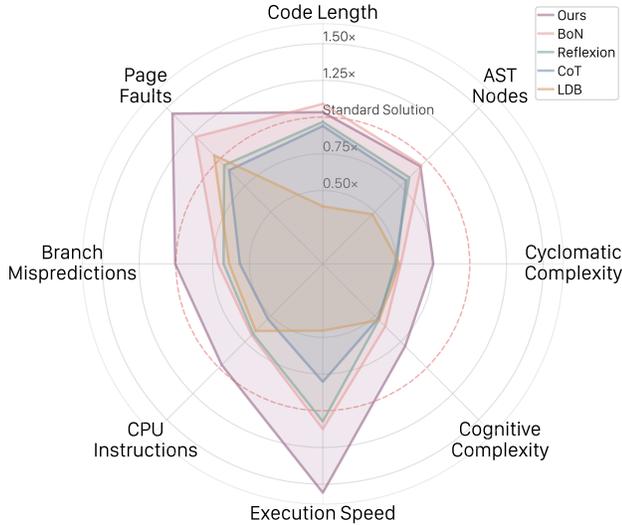


Figure 3: **Multi-dimensional Performance Analysis.** Metrics are normalized against the LBPP standard solutions (1.0 \times) and averaged across all backbone models. *Higher values indicate better performance.*

Table 3: **Ablation Study Results. - Execution:** Remove execution from our framework. **- Reasoning:** Remove reasoning process. Every metric is in percentage.

Method	Pass@1 \uparrow	Tests \uparrow	Valid \uparrow	Time \downarrow
ORPS	59.9	75.7	92.0	84.1
- Execution	43.8	56.4	72.8	200.5
- Reasoning	55.6	74.5	94.4	124.5

already exists in publicly available pre-training datasets.

4.3. Ablation Study

We conducted experiments on the challenging LBPP dataset using the Qwen-7B model to investigate the importance of two key components in the exploration process:

1. **Execution Outcomes.** Studies whether the model can access execution results to guide solution refinement. Without execution feedback, the model must rely solely on internal reasoning to assess correctness.
2. **Reasoning.** Investigates whether the model should explicitly perform reasoning before generating new code at each step. Without reasoning, the model directly generates new code based only on past code and execution results.

The results are presented in Table 3. When the model is unable to access execution outcomes during searching, Pass@1 decreases by 16.1%. *This highlights the critical role of environment feedback in guiding the model to gen-*

Table 4: **Analysis of Process Reward Model.** **Granularity** refers to the level of detail in the reward signal (line-level or outcome-level). **Train** indicates whether the process reward model requires training.

Methods		Pass@1 \uparrow	Tests \uparrow	Valid \uparrow	Time \downarrow
Granularity	Train				
Outcome	\checkmark	37.0	48.3	65.4	153.8
Line	\checkmark	32.1	43.9	59.3	153.4
Outcome	\times	59.9	75.7	92.6	89.1
Line	\times	38.3	52.8	70.4	123.7

erate correct solutions. Since LLMs struggle to accurately predict execution outcomes for a given piece of code (Jain et al., 2024), incorporating execution results ensures that the model benefits from concrete feedback.

Omitting reasoning during searching results in a 4.3% decrease in Pass@1. *Reasoning enables the model to iteratively refine its approach based on feedback, addressing issues that may not be resolved through execution feedback.*

4.4. Analysis of Process Reward Model

Our framework employs an implicit *process reward model* (PRM), which provides supervision signals during beam search without requiring additional training. To the best of our knowledge, no existing process supervision methods have been specifically designed for code generation. Most prior work on process supervision in mathematical reasoning (Luo et al., 2024) generates line-level reward signals and relies on explicitly trained PRMs. This motivates us to investigate two key questions: (1) Would adopting line-level rewards, similar to those used in mathematical reasoning tasks, improve performance? (2) Is an implicit PRM that does not require training better than an explicitly trained PRM?

To address these questions, we conduct experiments on the LBPP dataset using the Qwen-7B model. Similar to line-level rewards, we define the granularity of ORPS reward signals as *outcome-level*. For the line-level method, the model generates step-by-step thoughts for the coding problem, with numerical process rewards assigned to each step. The final code is then generated based on the best thought trace. For methods requiring explicit training, we randomly select half of the LBPP dataset as a training set to avoid data leakage. To simulate human-annotated process feedback, we filter data from GPT-4’s outputs.

Results in Table 4 confirm that our framework substantially outperforms the other three method, validating our design choices. Overall, outcome-level reward signals prove to be more effective than line-level signals. *Intuitively, line-level signals can only provide feedback for incomplete thought processes, which undoubtedly lack more information com-*

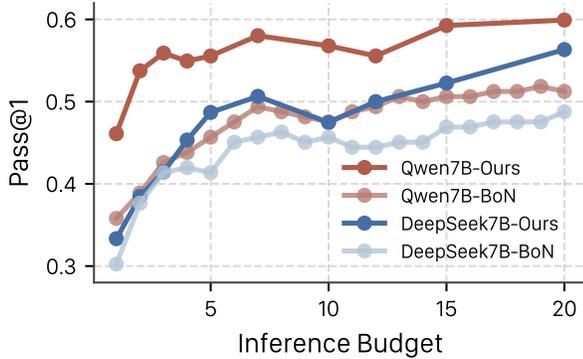


Figure 4: **Performance vs. Inference Budget.** The y-axis represents Pass@1 scores on LBPP. The x-axis represents the number of candidates generated during inference.

pared to the outcome-level.

Additionally, the implicit PRM shows greater effectiveness than the explicit PRM. This suggests that external process supervision feedback may not always be reliable. We consider that *LLMs already have strong self-reflection capabilities and only require execution outcomes to activate this ability*. This also indicates that spending extra data and time on training reward models might be unnecessary.

4.5. Scaling Analysis

In addition, we wanted to explore how the performance of ORPS changes as inference overhead increases. For comparison, we chose BoN as the baseline. This is because BoN allows easy control of inference overhead with linear growth. We conducted experiments on two models using the most challenging LBPP dataset. The results are shown in Figure 4. With the same model, ORPS improved much faster as inference overhead increased. This shows that ORPS has strong scaling potential. *It can effectively use more computational resources to improve reasoning*. In comparison, BoN showed slower improvements, suggesting it does not fully utilize the increased inference capacity.

4.6. Case Studies

We also analyzed the improvements of ORPS across different problem categories. As shown in Figure 5, on the competitive programming dataset LBPP, our method shows significant improvements over the CoT Baseline, especially in more difficult categories. For instance, in complex algorithmic tasks such as dynamic programming, loops, and graphs, our method correctly solves nearly twice as many problems as CoT. This further confirms that *high-quality intrinsic reasoning can help models avoid logical pitfalls when tackling difficult coding tasks*.

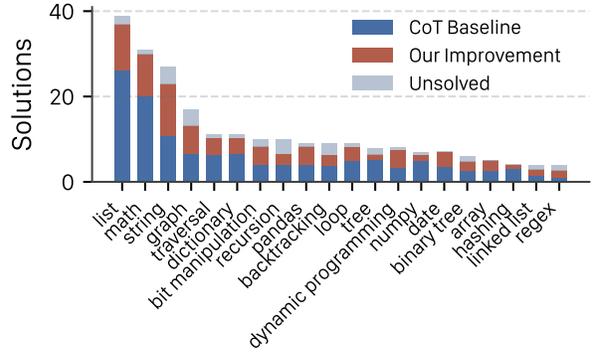


Figure 5: **Performance by Problem Class.** Top-20 problem classes in LBPP showing success rates and unsolved cases for ORPS vs baseline.

Through detailed case studies, we demonstrate how our framework enhances code generation by improving reasoning. As shown in Appendix C, the response generated by the traditional CoT method for the *Minimum Greatest Common Divisor* problem in LBPP demonstrates that while the model provides a detailed thought process during solution generation, the complexity of the task results in an imperfect code implementation. For instance, in CoT’s approach, the reliance on nested loops and pairwise GCD calculations introduces inefficiencies and fails to address scalability for larger datasets. Similarly, our method’s initial implementation demonstrates a lack of robustness in handling edge cases and unnecessary redundancies in subset formation.

However, ORPS achieves a more accurate solution through finer reasoning. The code initially generated by our model contains redundancies and erroneous logic. Nevertheless, with the feedback from the critic on the execution outcomes, the programmer successfully refines the code to reach a correct implementation. *This iterative process not only eliminates logical errors but also optimizes performance, demonstrating the advantage of integrating structured feedback into code generation*.

5. Conclusion

We introduce Outcome-Refining Process Supervision (ORPS), as a unified framework to bridge the gap between process supervision and outcome supervision through structured reasoning and execution-driven feedback. Our results reveal that structured reasoning concrete feedback signals are pivotal for solving complex programming tasks. ORPS proved effective even with smaller models, underscoring the importance of test-time scaling of reasoning over mere parameter scaling. Furthermore, our framework’s utilization of execution feedback eliminates the need for a trained PRM, making it a cost-efficient alternative.

These contributions highlight the potential of process supervision to enhance problem-solving abilities in LLMs. Future work could extend this framework to other domains requiring rigorous reasoning and verification. By bridging the gap between reasoning quality and execution fidelity, ORPS paves the way for more effective self-evolving LLMs.

Impact Statement

This work advances code generation capabilities in large language models through more efficient reasoning processes. While our primary focus is methodological—improving algorithmic problem-solving without costly reward models—we acknowledge broader implications common to code generation systems. Enhanced programming assistants could democratize software development but may also lower barriers for generating malicious code. The framework’s emphasis on code efficiency could reduce computational overhead in generated programs, though its environmental impact depends on deployment contexts.

These considerations reflect well-established societal trade-offs in AI-powered code tools rather than novel risks introduced by our approach. We encourage responsible deployment with standard safeguards against misuse, consistent with ethical practices for generative AI systems.

References

- Anil, R., Dai, A. M., Firat, O., Johnson, M., Lepikhin, D., Passos, A., Shakeri, S., Taropa, E., Bailey, P., Chen, Z., et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.
- Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- Ball, T. The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes*, 24(6):216–234, 1999.
- Bang, Y., Cahyawijaya, S., Lee, N., Dai, W., Su, D., Wilie, B., Lovenia, H., Ji, Z., Yu, T., Chung, W., et al. A multi-task, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. *arXiv preprint arXiv:2302.04023*, 2023.
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.
- Campbell, G. A. Cognitive complexity—a new way of measuring understandability. *SonarSource SA*, pp. 10, 2018.
- Chang, Y., Wang, X., Wang, J., Wu, Y., Zhu, K., Chen, H., Yang, L., Yi, X., Wang, C., Wang, Y., et al. A survey on evaluation of large language models. *arXiv preprint arXiv:2307.03109*, 2023.
- Chen, C.-C., Huang, H.-H., and Chen, H.-H. Evaluating the rationales of amateur investors. In *Proceedings of the Web Conference 2021*, pp. 3987–3998, 2021a.
- Chen, G., Liao, M., Li, C., and Fan, K. Alphamath almost zero: process supervision without process. *arXiv preprint arXiv:2405.03553*, 2024a.
- Chen, G. H., Chen, S., Liu, Z., Jiang, F., and Wang, B. Humans or llms as the judge? a study on judgement biases. *arXiv preprint arXiv:2402.10669*, 2024b.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021b.
- Chen, X., Lin, M., Schärli, N., and Zhou, D. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.
- Chen, Z., Zhao, Z., Zhu, Z., Zhang, R., Li, X., Raj, B., and Yao, H. Autoprml: Automating procedural supervision for multi-step reasoning via controllable question decomposition. *arXiv preprint arXiv:2402.11452*, 2024c.
- Chiang, W.-L., Li, Z., Lin, Z., Sheng, Y., Wu, Z., Zhang, H., Zheng, L., Zhuang, S., Zhuang, Y., Gonzalez, J. E., et al. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality. See <https://vicuna.lmsys.org> (accessed 14 April 2023), 2023.
- Clark, P., Cowhey, I., Etzioni, O., Khot, T., Sabharwal, A., Schoenick, C., and Tafjord, O. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.
- Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.
- Dai, N., Wu, Z., Zheng, R., Wei, Z., Shi, W., Jin, X., Liu, G., Dun, C., Huang, L., and Yan, L. Process supervision-guided policy optimization for code generation. *arXiv preprint arXiv:2410.17621*, 2024.
- Dodge, J., Ilharco, G., Schwartz, R., Farhadi, A., Hajishirzi, H., and Smith, N. Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping. *arXiv preprint arXiv:2002.06305*, 2020.

- Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
- Evans, J. S. B. In two minds: dual-process accounts of reasoning. *Trends in cognitive sciences*, 7(10):454–459, 2003.
- Freitag, M. and Al-Onaizan, Y. Beam search strategies for neural machine translation. *arXiv preprint arXiv:1702.01806*, 2017.
- Fu, J., Ng, S.-K., Jiang, Z., and Liu, P. Gptscore: Evaluate as you desire. *arXiv preprint arXiv:2302.04166*, 2023.
- Godbole, V., Dahl, G. E., Gilmer, J., Shalloe, C. J., and Nado, Z. Deep learning tuning playbook, 2023. URL http://github.com/google-research/tuning_playbook. Version 1.0.
- Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. *Deep learning*, volume 1. MIT Press, 2016.
- Gou, Z., Shao, Z., Gong, Y., Shen, Y., Yang, Y., Duan, N., and Chen, W. Critic: Large language models can self-correct with tool-interactive critiquing. *arXiv preprint arXiv:2305.11738*, 2023.
- Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y., et al. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.
- He, M., Shen, Y., Zhang, W., Tan, Z., and Lu, W. Advancing process verification for large language models via tree-based preference learning. *arXiv preprint arXiv:2407.00390*, 2024a.
- He, Y., Wang, H., Jiang, Z., Papangelis, A., and Zhao, H. Semi-supervised reward modeling via iterative self-training. *arXiv preprint arXiv:2409.06903*, 2024b.
- Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., and Steinhardt, J. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.
- Henighan, T., Kaplan, J., Katz, M., Chen, M., Hesse, C., Jackson, J., Jun, H., Brown, T. B., Dhariwal, P., Gray, S., et al. Scaling laws for autoregressive generative modeling. *arXiv preprint arXiv:2010.14701*, 2020.
- Hinton, G. E., Osindero, S., and Teh, Y. W. A fast learning algorithm for deep belief nets. *Neural Computation*, 18: 1527–1554, 2006.
- Hirschman, L. and Gaizauskas, R. Natural language question answering: the view from here. *natural language engineering*, 7(4):275–300, 2001.
- Hu, E. J., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W., et al. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2021.
- Hu, X., Gao, M., Hu, S., Zhang, Y., Chen, Y., Xu, T., and Wan, X. Are llm-based evaluators confusing nlg quality criteria? *arXiv preprint arXiv:2402.12055*, 2024.
- Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., and Zhou, D. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*, 2023.
- Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.
- Hurst, A., Lerer, A., Goucher, A. P., Perelman, A., Ramesh, A., Clark, A., Ostrow, A., Welihinda, A., Hayes, A., Radford, A., et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.
- Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.
- Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.
- Jiang, J., Chen, Z., Min, Y., Chen, J., Cheng, X., Wang, J., Tang, Y., Sun, H., Deng, J., Zhao, W. X., et al. Technical report: Enhancing llm reasoning with reward-guided tree search. *arXiv preprint arXiv:2411.11694*, 2024a.
- Jiang, J., Wang, F., Shen, J., Kim, S., and Kim, S. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024b.
- Jiao, F., Qin, C., Liu, Z., Chen, N. F., and Joty, S. Learning planning-based reasoning by trajectories collection and process reward synthesizing. *arXiv preprint arXiv:2402.00658*, 2024.
- Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

- Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.
- Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35: 22199–22213, 2022.
- Kwiatkowski, T., Palomaki, J., Redfield, O., Collins, M., Parikh, A., Alberti, C., Epstein, D., Polosukhin, I., Devlin, J., Lee, K., et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.
- Lewkowycz, A., Andreassen, A., Dohan, D., Dyer, E., Michalewski, H., Ramasesh, V., Slone, A., Anil, C., Schlag, I., Gutman-Solo, T., et al. Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems*, 35:3843–3857, 2022.
- Li, D., Jiang, B., Huang, L., Beigi, A., Zhao, C., Tan, Z., Bhattacharjee, A., Jiang, Y., Chen, C., Wu, T., et al. From generation to judgment: Opportunities and challenges of llm-as-a-judge. *arXiv preprint arXiv:2411.16594*, 2024.
- Li, Y. An open source data contamination report for llama series models. *arXiv preprint arXiv:2310.17589*, 2023.
- Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., and Cobbe, K. Let’s verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.
- Liu, Y., Chen, L., Wang, J., Mei, Q., and Xie, X. Meta semantic template for evaluation of large language models. *arXiv preprint arXiv:2310.01448*, 2023a.
- Liu, Y., Singh, A., Freeman, C. D., Co-Reyes, J. D., and Liu, P. J. Improving large language model fine-tuning for solving math problems. *arXiv preprint arXiv:2310.10047*, 2023b.
- Luo, L., Liu, Y., Liu, R., Phatale, S., Lara, H., Li, Y., Shu, L., Zhu, Y., Meng, L., Sun, J., et al. Improve mathematical reasoning in language models by automated process supervision. *arXiv preprint arXiv:2406.06592*, 2024.
- Ma, Q., Zhou, H., Liu, T., Yuan, J., Liu, P., You, Y., and Yang, H. Let’s reward step by step: Step-level reward model as the navigators for reasoning. *arXiv preprint arXiv:2310.10080*, 2023.
- Matton, A., Sherborne, T., Aumiller, D., Tommasone, E., Alizadeh, M., He, J., Ma, R., Voisin, M., Gilsenan-McMahon, E., and Gallé, M. On leakage of code generation evaluation datasets. *arXiv preprint arXiv:2407.07565*, 2024.
- Muñoz Barón, M., Wyrich, M., and Wagner, S. An empirical validation of cognitive complexity as a measure of source code understandability. In *Proceedings of the 14th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*, pp. 1–12, 2020.
- Nielson, F., Nielson, H. R., and Hankin, C. Principles of program analysis. 1999.
- OpenAI. Gpt-4 technical report, 2023.
- OpenAI. o1 system card. <https://cdn.openai.com/o1-system-card.pdf>, September 2024. Accessed: Dec 9, 2024.
- Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.
- Peng, B., Li, C., He, P., Galley, M., and Gao, J. Instruction tuning with gpt-4. *arXiv preprint arXiv:2304.03277*, 2023.
- Peng, K., Nisbett, R. E., and Wong, N. Y. Validity problems comparing values across cultures and possible solutions. *Psychological methods*, 2(4):329, 1997.
- Qin, Y., Li, X., Zou, H., Liu, Y., Xia, S., Huang, Z., Ye, Y., Yuan, W., Liu, H., Li, Y., et al. O1 replication journey: A strategic progress report—part 1. *arXiv preprint arXiv:2410.18982*, 2024.
- Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. Improving language understanding by generative pre-training. 2018.
- Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.
- Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2021.
- Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. Deep-speed: 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*, pp. 3505–3506, 2020.
- Riddell, M., Ni, A., and Cohan, A. Quantifying contamination in evaluating code generation capabilities of language models. *arXiv preprint arXiv:2403.04811*, 2024.
- Sainz, O., Campos, J. A., García-Ferrero, I., Etxaniz, J., de Lacalle, O. L., and Agirre, E. Nlp evaluation in trouble: On the need to measure llm data contamination for each benchmark. *arXiv preprint arXiv:2310.18018*, 2023.
- Setlur, A., Nagpal, C., Fisch, A., Geng, X., Eisenstein, J., Agarwal, R., Agarwal, A., Berant, J., and Kumar, A. Rewarding progress: Scaling automated process verifiers for llm reasoning. *arXiv preprint arXiv:2410.08146*, 2024.
- Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.
- Skalse, J., Howe, N., Krashennikov, D., and Krueger, D. Defining and characterizing reward gaming. *Advances in Neural Information Processing Systems*, 35:9460–9471, 2022.
- Snell, C., Lee, J., Xu, K., and Kumar, A. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.
- Stechly, K., Valmeekam, K., and Kambhampati, S. On the self-verification limitations of large language models on reasoning and planning tasks. *arXiv preprint arXiv:2402.08115*, 2024.
- Sun, C., Qiu, X., Xu, Y., and Huang, X. How to fine-tune bert for text classification? In *Chinese Computational Linguistics: 18th China National Conference, CCL 2019, Kunming, China, October 18–20, 2019, Proceedings 18*, pp. 194–206. Springer, 2019.
- Świechowski, M., Godlewski, K., Sawicki, B., and Mańdziuk, J. Monte carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, 2023.
- Thakur, A. S., Choudhary, K., Ramayapally, V. S., Vaidyanathan, S., and Hupkes, D. Judging the judges: Evaluating alignment and vulnerabilities in llms-as-judges. *arXiv preprint arXiv:2406.12624*, 2024.
- Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.
- Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.
- Tunstall, L., Von Werra, L., and Wolf, T. *Natural language processing with transformers*. ” O’Reilly Media, Inc.”, 2022.
- Uesato, J., Kushman, N., Kumar, R., Song, H. F., Siegel, N. Y., Wang, L., Creswell, A., Irving, G., and Higgins, I. Solving math word problems with process-based and outcome-based feedback. 2022.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *International Conference on Learning Representations*, 2018.
- Wang, A., Song, L., Tian, Y., Peng, B., Yu, D., Mi, H., Su, J., and Yu, D. Litesearch: Efficacious tree search for llm. *arXiv preprint arXiv:2407.00320*, 2024a.
- Wang, C., Liu, X., Yue, Y., Tang, X., Zhang, T., Jiayang, C., Yao, Y., Gao, W., Hu, X., Qi, Z., et al. Survey on factuality in large language models: Knowledge, retrieval and domain-specificity. *arXiv preprint arXiv:2310.07521*, 2023a.
- Wang, J., Fang, M., Wan, Z., Wen, M., Zhu, J., Liu, A., Gong, Z., Song, Y., Chen, L., Ni, L. M., et al. Openr: An open source framework for advanced reasoning with large language models. *arXiv preprint arXiv:2410.09671*, 2024b.
- Wang, P., Li, L., Chen, L., Cai, Z., Zhu, D., Lin, B., Cao, Y., Liu, Q., Liu, T., and Sui, Z. Large language models are not fair evaluators. *arXiv preprint arXiv:2305.17926*, 2023b.
- Wang, P., Li, L., Shao, Z., Xu, R., Dai, D., Li, Y., Chen, D., Wu, Y., and Sui, Z. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9426–9439, 2024c.
- Wang, Y., Kordi, Y., Mishra, S., Liu, A., Smith, N. A., Khashabi, D., and Hajishirzi, H. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*, 2022.

- Wang, Y., Yu, Z., Zeng, Z., Yang, L., Wang, C., Chen, H., Jiang, C., Xie, R., Wang, J., Xie, X., et al. Pandalm: An automatic evaluation benchmark for llm instruction tuning optimization. *arXiv preprint arXiv:2306.05087*, 2023c.
- Wang, Y., Yu, Z., Wang, J., Heng, Q., Chen, H., Ye, W., Xie, R., Xie, X., and Zhang, S. Exploring vision-language models for imbalanced learning. *International Journal of Computer Vision*, 132(1):224–237, 2024d.
- Wang, Z., Li, Y., Wu, Y., Luo, L., Hou, L., Yu, H., and Shang, J. Multi-step problem solving through a verifier: An empirical analysis on model-induced process supervision. *arXiv preprint arXiv:2402.02658*, 2024e.
- Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.
- Xie, R., Zeng, Z., Yu, Z., Gao, C., Zhang, S., and Ye, W. Codeshell technical report. *arXiv preprint arXiv:2403.15747*, 2024.
- Xiong, W., Song, Y., Zhao, X., Wu, W., Wang, X., Wang, K., Li, C., Peng, W., and Li, S. Watch every step! llm agent learning via iterative step-level process refinement. *arXiv preprint arXiv:2406.11176*, 2024.
- Yang, L., Zhang, S., Qin, L., Li, Y., Wang, Y., Liu, H., Wang, J., Xie, X., and Zhang, Y. Glue-x: Evaluating natural language understanding models from an out-of-distribution generalization perspective. *arXiv preprint arXiv:2211.08073*, 2022.
- Yang, L., Zhang, S., Yu, Z., Bao, G., Wang, Y., Wang, J., Xu, R., Ye, W., Xie, X., Chen, W., et al. Supervised knowledge makes large language models better in-context learners. *arXiv preprint arXiv:2312.15918*, 2023.
- Yao, W., Wang, Y., Yu, Z., Xie, R., Zhang, S., and Ye, W. Pure: Aligning llm via pluggable query reformulation for enhanced helpfulness. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 8721–8744, 2024.
- Yu, Z., Gao, C., Yao, W., Wang, Y., Ye, W., Wang, J., Xie, X., Zhang, Y., and Zhang, S. Kieval: A knowledge-grounded interactive evaluation framework for large language models. *arXiv preprint arXiv:2402.15043*, 2024a.
- Yu, Z., Gao, C., Yao, W., Wang, Y., Zeng, Z., Ye, W., Wang, J., Zhang, Y., and Zhang, S. Freeeval: A modular framework for trustworthy and efficient evaluation of large language models. *arXiv preprint arXiv:2404.06003*, 2024b.
- Zhang, D., Zhoubian, S., Hu, Z., Yue, Y., Dong, Y., and Tang, J. Rest-mcts*: Llm self-training via process reward guided tree search. *arXiv preprint arXiv:2406.03816*, 2024.
- Zhang, K., Li, Z., Li, J., Li, G., and Jin, Z. Self-edit: Fault-aware code editor for code generation. *arXiv preprint arXiv:2305.04087*, 2023.
- Zhang, T., Kishore, V., Wu, F., Weinberger, K. Q., and Artzi, Y. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*, 2019.
- Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, 2023.
- Zheng, Y., Zhang, R., Zhang, J., Ye, Y., Luo, Z., Feng, Z., and Ma, Y. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL <http://arxiv.org/abs/2403.13372>.
- Zhong, L., Wang, Z., and Shang, J. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*, 2024.

Table 5: **Performance Metrics Description.** Our evaluation framework uses both dynamic execution profiling and static code analysis metrics to comprehensively assess code quality and efficiency.

Category	Metric	Description
Dynamic Execution Profiling		
	Time Enabled	Total CPU time spent executing the code, measured in nanoseconds. Lower values indicate more efficient execution and better algorithmic optimization.
	Instruction Count	Number of CPU instructions executed during runtime. Reflects computational efficiency, with lower counts suggesting more optimized code paths and better algorithm implementation.
	Branch Misses	Frequency of incorrect branch predictions during execution. Lower values indicate better code predictability and CPU pipeline efficiency, resulting in faster execution times.
	Page Faults	Number of times the program needs to access virtual memory. Fewer page faults suggest better memory management and more efficient memory access patterns.
Static Analysis		
	Code Length	Total number of lines in the source code. Generally, shorter code length indicates more concise solutions while maintaining readability and functionality.
	AST Node Count	Number of nodes in the Abstract Syntax Tree. Measures structural complexity of the code, with fewer nodes suggesting simpler and more maintainable implementation.
	Cyclomatic Complexity	Quantifies the number of linearly independent paths through the code. Lower values indicate easier-to-maintain and test code, reducing potential bug sources.
	Cognitive Complexity	Measures how difficult the code is to understand, based on control flow structures and nesting. Lower scores suggest more readable and maintainable code that is easier to debug.

A. Experimental Setup and Hyperparameter Details

This appendix provides a comprehensive description of the experimental setup, encompassing the hyperparameters, software, and hardware configurations employed in this study.

A.1. Search Algorithm Hyperparameters (ORPS)

The following hyperparameters were used for the search algorithm in ORPS:

- **Search Depth (num_rounds):** 5. This parameter defines the maximum depth of the search tree, representing the number of iterative steps in the search process.
- **Beam Width (top_k):** 3. This parameter specifies the number of highest-scoring candidate solutions (traces) retained at each step of the beam search.
- **Expansion Factor (num_samples):** 20. This represents the number of new states (candidate solutions) explored from each state during the search process.
- **Process Reward Weight (α):** 0.5. This metric determines the proportion of the Process Reward in the step reward.
- **Outcome Reward Weight (β):** 0.5. This metric determines the proportion of the Outcome Reward in the step reward.

A.2. Inference Configuration

All inference experiments were conducted on a single machine using the FreeEval (Yu et al., 2024b) codebase, integrated with Hugging Face’s `text-generation-inference` toolkit for efficient model serving. The following inference

settings were applied:

- **Maximum Context Length (`max_tokens`):** 18,000 tokens. This parameter defines the maximum number of tokens allowed in the input sequence to the model.
- **Generated Tokens per Round:** 1,500 tokens. This specifies the number of new tokens generated by the model in each round of inference.

A.3. Execution Constraints

To ensure consistent and reproducible results, the following execution constraints were enforced during inference:

- **Timeout per Test Case:** 5 seconds. This limits the maximum execution time allowed for each test case.
- **Memory Limit:** 512 MB. This constraint restricts the maximum memory allocation permitted for each test case.
- **Maximum Test Cases per Problem:** 15. This sets an upper bound on the number of test cases evaluated for each problem.

A.4. Model Training Configuration

This section outlines the hyperparameters and settings used during the training phase of the model, which was pertinent to the analysis experiments (subsection 4.4). While ORPS itself does not require training, these details are provided for completeness and reproducibility.

- **Training Framework:** `llamafactory` (Zheng et al., 2024)
- **Optimization Framework:** DeepSpeed ZeRO3 (Rajbhandari et al., 2020) (Zero Redundancy Optimizer Stage 3). This enables efficient training of large models by partitioning optimizer states, gradients, and model parameters across data parallel processes.
- **Base Model:** `qwen-2.5-coder-7b-instruct`. This is the pre-trained language model upon which further training was conducted.
- **Batch Size per Device:** 2. This defines the number of training examples processed on each GPU before a gradient update step.
- **Gradient Accumulation Steps:** 4. This allows simulating a larger effective batch size by accumulating gradients over multiple forward and backward passes before updating model weights. The effective batch size is therefore 8 (2 per device * 4 steps).
- **Learning Rate:** 2×10^{-5} . This parameter controls the step size taken during gradient-based optimization.
- **Learning Rate Scheduler:** Cosine decay. This gradually reduces the learning rate over the course of training, following a cosine function.
- **Number of Training Epochs:** 2.0. This specifies the number of complete passes through the entire training dataset.
- **Maximum Sequence Length:** 16,384 tokens. This defines the maximum length of the input sequences during training.
- **Mixed Precision Training:** Enabled with `bfloat16` (Brain Floating Point 16-bit format). This accelerates training by performing some computations with reduced precision while maintaining model accuracy.

A.5. Hardware Environment

All experiments were performed on NVIDIA A800 GPUs, each equipped with 80GB of GPU memory.

CoT Output

Prob: Given an array of integers, split it into the minimum number of subsets such that the GCD of each subset is greater than 1.

Reasoning Process:

1. First, we need to understand what makes numbers group-able: - Numbers can be in the same subset if their GCD ≥ 1 - We want to minimize the number of subsets
 2. Key observations: - If $\text{GCD}(a,b) \geq 1$ and $\text{GCD}(b,c) \geq 1$, a,b,c can potentially be in the same group - This suggests a graph-like structure where numbers are connected if their GCD ≥ 1
 3. Solution approach: - Treat numbers as nodes in a graph - Connect numbers with edges if their GCD ≥ 1 - Find connected components in this graph - Each component represents one subset
 4. Implementation strategy: - Use Union-Find data structure for efficient component tracking - Calculate GCD using Euclidean algorithm - Final answer is the number of connected components
- Generated Solution:

```

1 def gcd(a, b):
2     while b:
3         a, b = b, a % b
4     return a
5
6 def minGroups(nums):
7     n = len(nums)
8     parent = list(range(n))
9
10    def find(x):
11        if parent[x] != x:
12            parent[x] = find(parent[x])
13        return parent[x]
14
15    def union(x, y):
16        parent[find(x)] = find(y)
17
18    for i in range(n):
19        for j in range(i + 1, n):
20            if gcd(nums[i], nums[j]) > 1:
21                union(i, j)
22
23    return len(set(find(i) for i in range(n)))

```

B. AI Usage in Code Development

During the development of ORPS and the design of its experiments, LLMs were employed to assist with coding. All AI-assisted code were reviewed and refined by the authors to ensure correctness and alignment with the research goals.

C. Example Model Outputs

To illustrate the effectiveness of our approach, we present a representative example from the LBPP dataset (Problem #101) that demonstrates how our method handles complex logical reasoning tasks. The original prompt of this problem is:

You are given an array of integers, each of which is greater than 1. This array needs to be split into subsets such that the greatest common divisor within each subset is greater than 1. Write a Python program to return the minimum number of subsets required to create such a split.

ORPS Programmer Output

=== Programmer Thoughts === # The problem requires splitting an array into subsets such that the greatest common divisor (GCD) within each subset is greater than 1. To achieve this, we can use a greedy algorithm. The idea is to iterate through the array and add each element to the existing subset if their GCD is greater than 1. If the GCD is 1, we start a new subset.

We need to implement a function that calculates the GCD of two numbers. Then, we can use this function to determine if an element can be added to an existing subset or if we need to start a new subset.

We also need to ensure that our solution is efficient. The time complexity of the GCD calculation can be optimized using the Euclidean algorithm, which has a time complexity of $O(\log(\min(a, b)))$.

=== Solution ===

```

1 import math
2
3 def gcd(a, b):
4     """
5     Calculate the greatest common divisor of two numbers using the Euclidean algorithm.
6     """
7     while b:
8         a, b = b, a % b
9     return a
10
11 def min_g_c_d_subsets(nums):
12     """
13     Split the array into subsets such that the greatest common divisor within
14     each subset is greater than 1. Return the minimum number of subsets required.
15     """
16     subsets = []
17     for num in nums:
18         added = False
19         for subset in subsets:
20             if gcd(num, subset[-1]) > 1:
21                 subset.append(num)
22                 added = True
23                 break
24             if not added:
25                 subsets.append([num])
26     return len(subsets)

```

ORPS Critic Feedback

=== Critic Thoughts === The code provided attempts to solve the problem by iterating through each number and adding it to an existing subset if the GCD with the last element of the subset is greater than 1. If no such subset exists, a new subset is created. This approach ensures that each subset has a GCD greater than 1. However, there are a few issues and improvements that can be made:

1. **Test Case Failures**: The test cases for indices 5, 8, and 10 fail. This indicates that the current implementation does not correctly handle certain scenarios where numbers with no common divisor greater than 1 are present.
2. **Performance**: The code has a higher cognitive complexity, which means it is harder to understand and maintain. The use of nested loops and the nested if-else structure can be optimized.
3. **Efficiency**: The current approach has a time complexity of $O(n^2)$ due to the nested loops. This can be optimized to $O(n \log n)$ by sorting the numbers first and then using a single pass to form subsets.
4. **Code Clarity**: The code can be made more readable by breaking down the logic into smaller functions and adding comments.
5. **Handling Edge Cases**: The code does not handle edge cases such as an empty list or a list with a single element.

=== Score === \$\$\$

ORPS Programmer Improvement

=== Programmer Thoughts === # The feedback indicates that the current implementation has several issues, including failing test cases and high cognitive complexity. We need to address these issues by improving the logic and ensuring that all edge cases are handled correctly.

Key Improvements:

1. **Handling Edge Cases**: Ensure that the function handles an empty list or a list with a single element.
2. **Efficiency**: Optimize the approach to achieve a time complexity of $O(n \log n)$ by sorting the numbers first and then using a single pass to form subsets.
3. **Code Clarity**: Break down the logic into smaller functions and add comments for better readability.

Approach:

1. **Sort the Array**: Sorting the array ensures that numbers with a common divisor are adjacent, making it easier to form subsets.
2. **Single Pass to Form Subsets**: Iterate through the sorted array and add each number to the current subset if the GCD with the last element of the subset is greater than 1. If not, start a new subset.
3. **Edge Case Handling**: If the list is empty, return 0. If the list has a single element, return 1.

=== Solution ===

```

1 import math
2
3 def gcd(a, b):
4     """Calculate the greatest common divisor of two numbers
5     using the Euclidean algorithm."""
6     while b:
7         a, b = b, a % b
8     return a
9
10 def min_g_c_d_subsets(nums):
11     """
12     Split the array into subsets such that the greatest common divisor
13     within each subset is greater than 1.
14     Return the minimum number of subsets required.
15
16     :param nums: List of integers greater than 1
17     :return: Minimum number of subsets required
18     """
19     if not nums:
20         return 0
21     if len(nums) == 1:
22         return 1
23
24     # Sort the array to ensure numbers with a common divisor are adjacent
25     nums.sort()
26
27     subsets = []
28     for num in nums:
29         added = False
30         for subset in subsets:
31             if gcd(num, subset[-1]) > 1:
32                 subset.append(num)
33                 added = True
34                 break
35         if not added:
36             subsets.append([num])
37
38     return len(subsets)

```

D. Impact of Optimizing Different Metrics on Code Quality

In ORPS, we integrate multiple evaluation metrics to guide the model’s reasoning and code generation. These metrics include both **static analysis** (e.g., AST nodes, cyclomatic complexity) and **dynamic execution profiling** (e.g., execution speed, CPU instruction count, branch mispredictions). To better understand their individual contributions, we conduct an ablation study where each metric is optimized in isolation.

D.1. Experimental Setup

To isolate the effect of each metric, we conduct a series of ablation experiments. In each experiment, the model receives **rewards only from a single metric**, while all other evaluation criteria remain unchanged. Specifically, we consider the following setups:

- **+AST Nodes**: Encourages structurally simpler code by minimizing the number of AST nodes.
- **+Cyclomatic Complexity**: Penalizes excessive branching and loop structures to improve maintainability.
- **+Cognitive Complexity**: Rewards code that is easier to understand based on nested structures and control flow.
- **+Execution Speed**: Optimizes for faster execution while maintaining correctness.
- **+CPU Instructions**: Minimizes the number of CPU instructions executed.
- **+Branch Mispredictions**: Encourages predictability to improve processor efficiency.
- **+Page Faults**: Reduces memory access overhead for better performance.
- **+All Metrics**: Incorporates all the above metrics into a single optimization objective.

D.2. Results and Discussion

As shown in [Figure 6](#), optimizing for a single metric significantly improves performance in that specific aspect, yet comes at a severe cost to other dimensions of code quality. This phenomenon suggests that the model falls into a form of **local metric optimality**, where it overfits to the given reward signal while neglecting other critical properties of high-quality code.

More specifically, we observe that optimizing for **static analysis metrics** (e.g., AST nodes, cyclomatic complexity) often leads to a sharp decline in **dynamic execution metrics** (e.g., execution speed, CPU instructions). For instance, minimizing cyclomatic complexity encourages structurally simpler code, yet it may suppress more efficient algorithmic choices that involve loops and conditionals. Conversely, optimizing for execution speed often results in obfuscated or redundant code, as the model prioritizes raw performance over maintainability.

These observations highlight an inherent challenge in code optimization: static structure and dynamic efficiency often conflict when optimized in isolation. This suggests that achieving well-balanced code quality requires a **multi-objective optimization** strategy rather than single-metric reinforcement. Traditional reward models struggle in such scenarios, as they often assume reward signals are aligned across different dimensions. However, our results indicate that code generation requires more nuanced supervision—one that dynamically balances trade-offs between readability, maintainability, and execution efficiency.

Furthermore, the steep performance drop in non-optimized metrics suggests that **reward sparsity** is a critical issue in single-metric training. Since the model receives no information about other quality dimensions, it fails to generalize improvements beyond the specific reward it observes. This contrasts with human programming intuition, where engineers naturally balance multiple objectives, such as runtime efficiency, readability, and modularity. Future work could explore techniques like **adaptive reward scaling**, where the model dynamically adjusts its focus based on real-time trade-offs rather than rigid metric-specific optimization.

	Static Analysis Improvements				Dynamic Analysis Improvements			
No Metrics (Baseline)	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
+Code Length	+85.2%	+32.5%	-11.9%	+160.1%	-113.5%	-54.5%	-57.4%	+6.3%
+AST Nodes	+43.4%	+19.3%	-8.1%	+146.6%	-217.2%	-134.3%	-186.5%	-161.8%
+Cyclomatic Complexity	+43.1%	+43.9%	+178.6%	+146.6%	-22.1%	+48.7%	+27.1%	-18.8%
+Cognitive Complexity	+62.8%	+40.8%	+126.3%	+203.3%	-67.7%	+13.9%	-14.3%	-24.6%
+Execution Speed	+3.1%	-11.5%	+23.7%	+33.2%	-23.6%	+46.0%	+16.9%	+13.6%
+CPU Instructions	-43.8%	-61.0%	-49.2%	-25.3%	-49.9%	+22.5%	-12.9%	-5.0%
+Branch Mispredictions	-24.9%	-52.6%	-31.0%	-8.8%	-30.9%	+34.0%	+7.2%	0.0%
+Page Faults	+54.0%	+49.4%	+119.7%	+130.7%	+10.7%	+78.7%	+58.2%	+71.4%
+All Metrics	+44.0%	+44.4%	+96.8%	+82.4%	+48.9%	+86.6%	+81.5%	+35.7%
	Code Length	AST Nodes	Cyclomatic Complexity	Cognitive Complexity	Execution Speed	CPU Instructions	Branch Mispredictions	Page Faults

Figure 6: **Impact of single-metric optimization on code quality.** Each value represents the difference from the baseline. Optimizing for a single metric significantly improves performance in that dimension but leads to severe degradation in others.

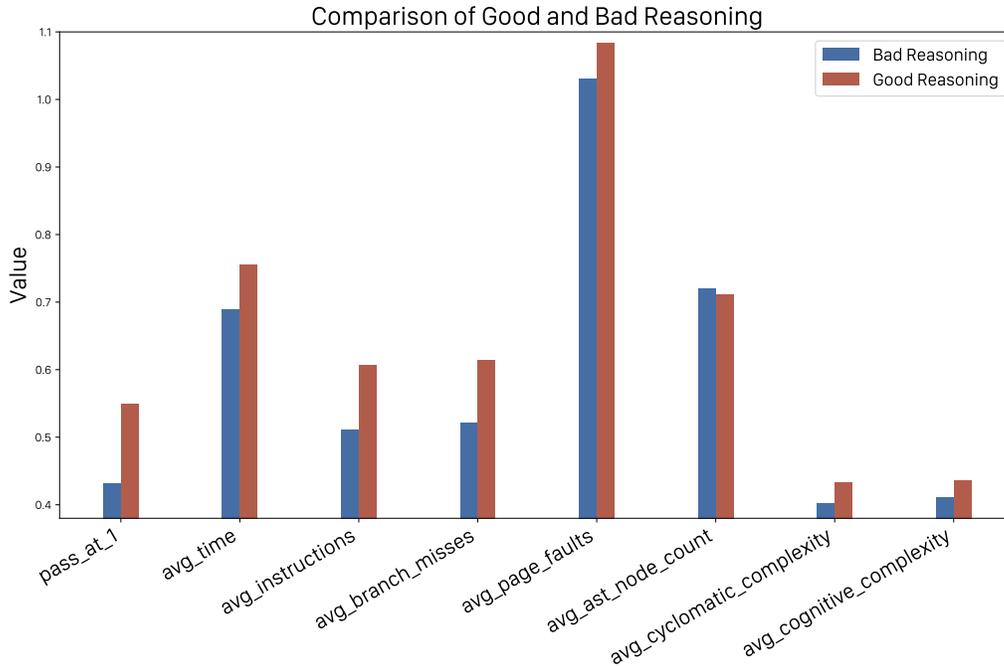


Figure 7: **Impact of reasoning quality on code quality.** Each metric is normalized, where higher values indicate better performance. The red bars represent the code quality when generated from higher-quality reasoning, while the blue bars correspond to lower-quality reasoning. The results suggest that better reasoning significantly improves execution-related metrics while having a limited effect on structural complexity.

E. Relationship Between Reasoning Quality and Code Quality

Our method is based on a core hypothesis: **higher-quality reasoning leads to higher-quality code**. To validate this assumption, we conducted a simple motivation experiment.

E.1. Experimental Setup

We first prompted GPT-4o to generate two different reasoning chains for each problem in the LBPP dataset, explicitly ensuring a clear quality difference between them. To verify that the generated reasoning indeed exhibited significant quality differences, we used GPT-4o again to reassess and confirm their relative quality.

Next, we concatenated each reasoning chain with the original problem description and fed them separately into GPT-4o to generate code solutions. Finally, we evaluated the quality of the generated code across multiple metrics.

E.2. Results and Analysis

As shown in Figure 7, all metrics are normalized, where higher values indicate better performance. The red bars represent the quality of code generated from higher-quality reasoning, while the blue bars correspond to the lower-quality reasoning.

From the results, we observe that **better reasoning generally leads to better overall code quality**, particularly in **dynamic execution metrics**. This suggests that high-quality reasoning not only improves correctness but also enhances execution efficiency, possibly by guiding the model to generate more optimal algorithmic structures.

Interestingly, the complexity of the generated code does not increase significantly. One possible explanation is that **more detailed and higher-quality reasoning naturally guides the model toward generating more sophisticated solutions that handle a wider range of scenarios**, rather than simply making the code structurally more complex.

These findings reinforce the importance of reasoning supervision in code generation. A strong reasoning framework helps models not only arrive at correct solutions but also optimize execution efficiency without unnecessarily inflating complexity.