MUPCODER: MITIGATING REWARD SPARSITY IN CODE GENERATION VIA MULTI-PERSPECTIVE TOKEN LEVEL REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

The demands of syntactic correctness and logical precision in code generation remain challenging for off-the-shelf large language models, motivating post-training alignment as a common approach. However, current approaches face critical constraints: Supervised Fine-Tuning (SFT) is prone to in-distribution overfitting and struggles with out-of-distribution generalization, while Reinforcement Learning (RL) suffers from reward sparsity. Therefore, we propose MupCoder (**Mu**lti-**p**erspective **Coder**) and a token-level RL framework that provides dense process supervision without requiring auxiliary reward models. Rather than relying solely on sparse outcome feedback, MupCoder equips the model with multi-perspective hints, such as reference demonstrations and test cases. Conditioned on these contexts, the model acts as its own teacher via on-policy self-distillation, deriving dense token-level rewards from the sampled-token log-probability gap between the standard student branch and hint-conditioned teacher branches. To capture long-horizon dependencies in code generation, we further accumulate these dense signals through a future-aware discounted return, which propagates downstream alignment information to earlier tokens. This dense shaping signal is combined with sparse execution feedback to anchor functional correctness. Experiments on multiple code generation benchmarks show that MupCoder consistently improves generation accuracy over representative baselines.

1 INTRODUCTION

Large Language Models (LLMs) have become increasingly important in software engineering, establishing code generation as a pivotal capability for enhancing developer productivity and automating complex programming tasks (Jiang et al., 2026; Ziegler et al., 2022). From synthesizing routine code to solving complex algorithmic challenges (Chen et al., 2021; Austin et al., 2021), the ability to generate logically correct and syntactically sound code is increasingly critical in both academic research and industrial applications. As the demand for reliable automated coding assistants grows, ensuring that these models can consistently produce high-quality code in diverse scenarios has become a central focus of current research (Hou et al., 2024).

However, reliable code generation remains challenging (Jimenez et al., 2023; Zhang et al., 2025; 2026), motivating post-training methods such as Supervised Fine-Tuning (SFT) and Reinforcement Learning (RL) (Ouyang et al., 2022; Sutton et al., 1998). While SFT is a standard approach, it frequently suffers from limited generalization to out-of-distribution inputs (Bengio et al., 2015). To overcome this, RL offers an exploratory mechanism to optimize directly for specific outcome objectives, but is notoriously hindered by reward sparsity. As illustrated in Fig. 1, standard RL like Group Relative Policy Optimization (GRPO) (Shao et al., 2024) assigns a single uniform reward to every token in the code. However, in the domain of code generation, functional success often depends on a few critical tokens (Lu et al., 2025). Relying solely on such uniform sequence-level feedback fails to differentiate their distinct importance, providing insufficient guidance. This motivates the use of finer-grained token-level supervision that can effectively bridge the gap between local syntactic validity and global execution correctness.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

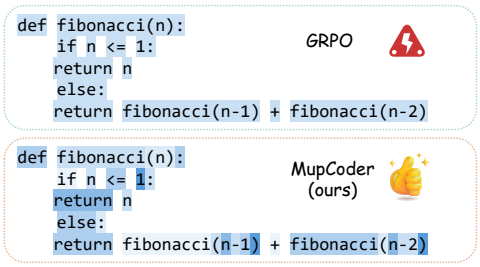

Figure 1: Comparison of reward granularity between GRPO (top) and MupCoder (bottom). The color intensity illustrates the magnitude of the assigned reward. While GRPO relies on a single outcome reward applied uniformly to the entire sequence, our approach assigns a distinct and fine-grained reward signal to each token.

Therefore, we propose the MupCoder (**Mu**lti-**p**erspective **Coder**), a novel token-level RL framework tailored for code generation, equipped with a hybrid objective that cohesively integrates multi-perspective, token-level dense supervision with sparse outcome-based rewards. Specifically, inspired by On-Policy Distillation (OPD) (Lu & Lab, 2025), which utilizes the token-level KL divergence between a teacher and a student as a dense signal (detailed in Sec. 2.1), we eliminate the need for an external teacher model. Instead, we provide the student with multi-perspective hints (*e.g.,* demonstration, pseudocode, test cases) as hinted contexts, enabling it to be a self-teacher and self-distill knowledge across diverse reasoning paths. Furthermore, recognizing strict sequential dependencies in coding, where an early erroneous token can affect subsequent generation and execution outcomes (Lu et al., 2025; Dou et al., 2024), we design a future-aware reward mechanism. This ensures that the dense reward for each token encapsulates its cascading impact on subsequent generation steps. Finally, by hybridizing this future-aware dense signal with verifiable outcome feedback, MupCoder provides precise, exploratory, and globally aligned guidance throughout the generation process.

We conduct extensive experiments on widely adopted code generation benchmarks, including MBPP (Austin et al., 2021), HumanEval (Chen et al., 2021), and the challenging LiveCodeBench (Jain et al., 2024). Our results show that MupCoder improves over representative post-training and process-supervision baselines, achieving higher generation accuracy across the evaluated benchmarks. Furthermore, the integration of dense token-level signals facilitates faster convergence compared to traditional sparse-reward RL methods, validating the effectiveness of MupCoder in mitigating the reward sparsity bottleneck while maintaining high-quality code generation.

In summary, our primary contributions are as follows:

- We propose MupCoder and a token-level RL framework for code generation that mitigates reward sparsity by integrating dense token-level shaping signals with sparse execution-based outcome rewards, connecting fine-grained generation decisions with global functional correctness.
- We introduce a multi-perspective self-distillation mechanism that derives token-level rewards from the log-probability gap between the standard branch and hint-conditioned branches. Besides, we use a discounted return to propagate downstream alignment signals to earlier tokens.
- We demonstrate consistent empirical gains across MBPP, HumanEval, and LiveCodeBench. MupCoder outperforms post-training and process-supervision baselines in generation accuracy, while showing faster convergence under the same backbone model.

2 Method

Fig. 2 illustrates the overall architecture of the MupCoder framework, which mainly consists of two components: multi-perspective hint construction and token-level reward modeling.

2.1 Preliminaries

We first briefly review on-policy distillation (OPD), which serves as the basis of our token-level reward construction. OPD trains on trajectories sampled from the student policy, while using a teacher model to provide fine-grained token-level supervision (Lu & Lab, 2025).

Formally, for a prompt x drawn from the training set, the student policy π_θ first samples a trajectory $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_T) \sim \pi_\theta(\cdot \mid x)$. At each step t, both the student and a teacher model π_T evaluate the next-token distribution conditioned on the student-generated prefix $\hat{y}_{<t}$:

$$p_t(\hat{y}_t) \triangleq \pi_\theta(\hat{y}_t \mid x, \hat{y}_{<t}), \quad q_t(\hat{y}_t) \triangleq \pi_T(\hat{y}_t \mid x, \hat{y}_{<t}). \tag{1}$$

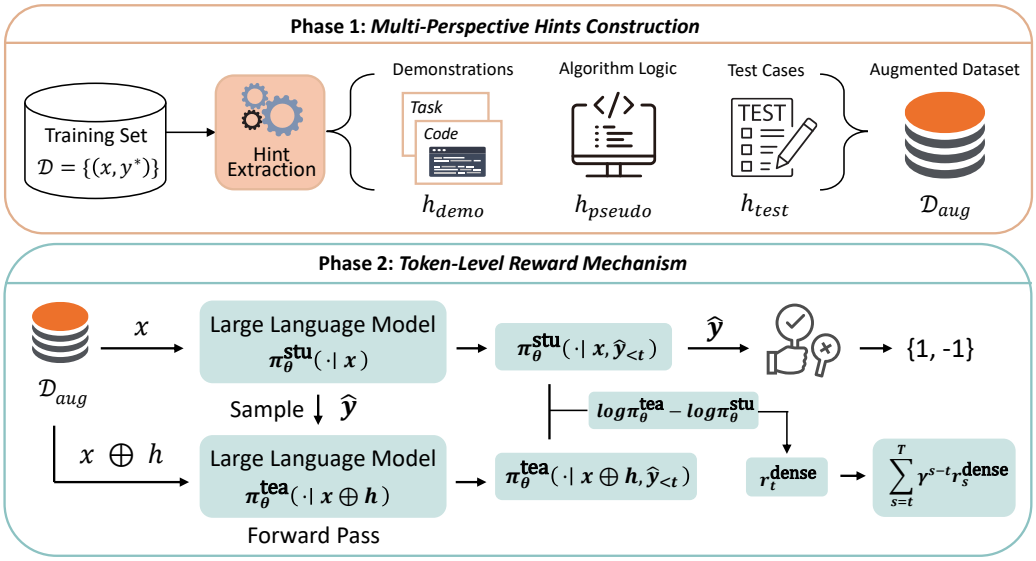

Figure 2: The overall architecture of the MupCoder framework. Phase 1 outlines the multi-perspective hint extraction process, where three types of hints $h \in \mathcal{H}$ are derived from the problem description x and its corresponding reference solution y^*. Phase 2 illustrates the token-level reward design workflow. For the sake of visual simplicity, the diagram in Phase 2 depicts the mechanism using a single hint h, which serves to represent the averaged effect across all three hint modalities.

The standard objective minimizes the reverse KL divergence over these student rollouts:

$$\mathcal{L}_{\text{OPD}}(\theta) = \mathbb{E}_{x \sim \mathcal{D}_x}\big[D_{\text{KL}}(\pi_\theta(\cdot \mid x) \,\|\, \pi_T(\cdot \mid x))\big]. \tag{2}$$

In practice, we use the sampled-token $\ell_t = \log p_t(\hat{y}_t) - \log q_t(\hat{y}_t)$ to estimate the reverse KL divergence. The key point is that every position now carries its own supervision. Instead of a sparse reward at the end, the KL term gives the student immediate, local feedback on where it diverges from the teacher.

Importantly, recent studies indicate that the performance gains of OPD do not stem merely from the parameter scale of the teacher. Li et al. (2026) demonstrate that simply scaling up a teacher trained under similar paradigms yields marginal benefits if it lacks distinct, complementary knowledge. The efficacy of token-level supervision relies intrinsically on the information content of the teacher's distribution, rather than its raw size. Furthermore, maintaining a separate, massive teacher model incurs prohibitive computational costs. Consequently, the critical challenge is how to construct a teacher distribution rich in transferable information without relying on an external model.

2.2 Multi-Perspective Hints Construction

Instead of relying on a massive external teacher model, our method explicitly constructs a richer context distribution for self-teaching. Code generation tasks naturally facilitate this, as programming problems can be decomposed into structured modalities—such as algorithmic logic, design patterns, and input-output constraints—which can be systematically extracted to enrich the teacher's perspective. These hints are used only to construct training-side supervision, not in evaluation.

A naive approach to leverage this available information would be to simply condition the teacher branch directly and only on the ground-truth solution (Zhao et al., 2026). However, this is deeply problematic. Code generation is inherently multi-solution; exposing the teacher to the exact reference code collapses its distribution onto a single implementation path, which strips the student of its exploratory freedom (Jang et al., 2023; Yang et al., 2026). To preserve generation diversity while still providing robust guidance, we employ three types of multi-perspective hints.

Demonstration (h_{demo}). We retrieve a similar problem from the training set together with its complete solution, excluding the target problem itself (see App. D.1 for retrieval details). This problem–solution pair serves as a worked example, exposing potentially reusable strategies for organizing control flow, and decomposing the task. Prior work has shown that appropriately selected in-context demonstrations can improve code generation by surfacing relevant problem-solving patterns (Gao et al., 2023).

Student Input $\pi_\theta(\cdot \mid x)$

```
[Problem Description] {x}
[Task] Generate a Python solution for the
above problem.
```

Self-Teacher Input $\pi_\theta(\cdot \mid x \oplus h)$

```
[Problem Description] {x}
[Hint: {Hint Type}] {h}
[Task] Generate a Python solution for the
above problem. Please do not reveal or copy
the hint information in the solution.
```

Figure 3: Training prompts for the student and self-teacher. Hints h are provided only to the teacher branch during training.

Pseudocode (h_{pseudo}). To expose the target solution's algorithmic structure while reducing language specific syntax, we convert the ground-truth code into pseudocode using the open source `PseudoGen` framework (Fudaba et al., 2015) (see App. D.2 for construction details). The resulting hint describes the main computational steps, including conditional branches, iterative updates, and operations on data structures. Although it retains solution-derived information, this representation guides the teacher toward the intended correct algorithm.

Test Cases (h_{test}). We provide public input output examples from the problem's test suite as an additional teacher-side hint. These examples clarify expected behavior and, when available, expose boundary conditions or special cases that are under-specified in the problem description. Unlike demonstration and pseudocode, test cases describe what the program should do without prescribing a particular implementation. Their role as contextual guidance complements the outcome reward, which directly checks whether the student's completed program passes the training tests.

2.3 Token-Level Reward Mechanism

Building on the multi-perspective hints constructed in Sec. 2.2, MupCoder combines contextual self-distillation with execution feedback to guide policy optimization. Following the on-policy distillation framework in Sec. 2.1, we retain student-generated rollouts and replace the external teacher with hint-conditioned branches of the same policy. Below, we define the dense token-level reward, the sparse outcome reward, and their integration through a future-aware return.

Dense Token-Level Reward. Let $\mathcal{H} = \{h_{\text{demo}}, h_{\text{pseudo}}, h_{\text{test}}\}$ denote the three complementary hints associated with each training problem. Given a student rollout $\hat{y} \sim \pi_\theta(\cdot \mid x)$, the student and teacher branches evaluate each sampled token on the same prefix:

$$p_t(\hat{y}_t) \triangleq \pi_\theta(\hat{y}_t \mid x, \hat{y}_{<t}), \quad q_t^{(h)}(\hat{y}_t) \triangleq \pi_\theta(\hat{y}_t \mid x \oplus h, \hat{y}_{<t}), \quad h \in \mathcal{H}. \tag{3}$$

Both branches share parameters and differ only in their conditioning contexts. We use their sampled-token log-probability gap, $\ell_t^{(h)} = \log p_t(\hat{y}_t) - \log q_t^{(h)}(\hat{y}_t)$, as an estimate of the reverse-KL contribution, and average its negative across hints to obtain:

$$r_t^{\text{dense}} = \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \left(\log q_t^{(h)}(\hat{y}_t) - \log p_t(\hat{y}_t)\right) = -\frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \ell_t^{(h)}. \tag{4}$$

This reward combines guidance from multiple teacher contexts: it is positive when the average teacher log-probability exceeds the student's and negative otherwise. It therefore encourages alignment with hint-informed predictions along the student's own generation path. Training prompts are shown in Fig. 3, and a diagnostic evaluation of hint informativeness is provided in App. I.

Sparse Outcome Reward. The dense reward measures distributional alignment rather than functional correctness. To provide direct behavioral feedback, we execute the completed program against the training test suite and assign a binary outcome reward:

$$R^{\text{outcome}} = \begin{cases} 1, & \text{if the generated program passes all training tests,} \\ -1, & \text{otherwise.} \end{cases} \tag{5}$$

Unlike the token-level dense signal, this reward evaluates the completed program as a whole. It complements contextual guidance with a verifiable objective that favors programs satisfying the tested functional requirements.

Future-Aware Dense Sparse Integration. Code generation involves strong prefix dependencies: an early incorrect variable or API choice can affect subsequent decisions and may only be exposed during execution (Lu et al., 2025; Le et al., 2022). To incorporate downstream alignment information, we accumulate dense rewards into a discounted return:

$$G_t^{\text{dense}} = \sum_{s=t}^{T} \gamma^{s-t} r_s^{\text{dense}}, \tag{6}$$

where $\gamma \in [0, 1]$ controls the contribution of future signals. We then combine this return with the execution-based outcome reward:

$$G_t = \underbrace{\sum_{s=t}^{T} \gamma^{s-t} r_s^{\text{dense}}}_{\text{future-aware dense signal}} + \underbrace{\lambda R^{\text{outcome}}}_{\text{execution-based outcome signal}}, \tag{7}$$

where $\lambda > 0$ balances the two sources of supervision. The resulting hybrid return provides each token with both local and downstream alignment signals, together with feedback on the completed program's execution correctness.

Policy optimization. With the hybrid return defined, we optimize the student policy with a GRPO-style clipped policy objective (Shao et al., 2024). While standard GRPO relies on group sampling to compute relative advantages, our dense reward directly captures the student–teacher log-probability gap and serves as an intrinsic token-level advantage. This allows us to use a rollout size of 1 (Zeng et al., 2026), reducing memory and computational overhead.

We optimize the clipped surrogate objective:

$$\mathcal{J}(\theta) = \mathbb{E}_{x \sim \mathcal{D}_x,\ \hat{y} \sim \pi_{\theta_{\text{old}}}(\cdot|x)} \left[\sum_{t=1}^{T} \min \left(\rho_t(\theta) \hat{A}_t,\ \text{clip}\big(\rho_t(\theta), 1-\epsilon, 1+\epsilon\big) \hat{A}_t \right) \right], \tag{8}$$

where $\rho_t(\theta) = \frac{\pi_\theta(\hat{y}_t | x, \hat{y}_{<t})}{\pi_{\theta_{\text{old}}}(\hat{y}_t | x, \hat{y}_{<t})}$ denotes the probability ratio between the current and rollout policies, ϵ is the clipping hyperparameter, and $\hat{A}_t = \text{sg}(G_t)$ serves as the token-level advantage.

At each training iteration, we use the current policy as $\pi_{\theta_{\text{old}}}$ and sample an on-policy trajectory $\hat{y}$ from the original prompt x. Using the same snapshot, we compute both the student and hint-conditioned teacher log-probabilities on the shared prefix $\hat{y}_{<t}$, construct the hybrid return G_t, and detach it before updating π_θ. Thus, $\text{sg}(G_t)$ serves as a fixed token-level weight in the clipped policy objective, without backpropagating through the reward computation.

3 Experiments

3.1 Experimental Setup

Training and Evaluation Datasets. We construct a training set of `79,311` instances by integrating MBPP train split (Austin et al., 2021), AceCoder (Zeng et al., 2025), and a subset of interview difficulty problems from APPS (Hendrycks et al., 2021). Each instance consists of a problem description, a reference solution, and corresponding test cases. To ensure data quality, we apply standard filtering protocols: (1) discarding syntactically invalid code, (2) removing tasks with invalid test suites, (3) verifying that all reference solutions pass their respective tests, and (4) deduplicating the merged corpus and removing identified overlaps with the evaluation benchmarks. We evaluate on HumanEval (Chen et al., 2021), MBPP, and LiveCodeBench-V6 (Jain et al., 2024).

Table 1: Performance comparison of MupCoder and baselines on HumanEval+, MBPP+, and LiveCodeBench-V6 benchmarks across multiple model scales. Best results in each model scale are **bolded**, second best are underlined.

Method	HumanEval+		MBPP+ (sanitized)		LiveCodeBench-V6		Avg. Pass@1	Avg. Pass@10
	Pass@1	Pass@10	Pass@1	Pass@10	Pass@1	Pass@10		
Qwen2.5-Coder-1.5B								
Instruct	62.7	73.3	56.0	70.2	5.8	10.8	41.5	51.4
+ SFT	66.2	79.1	61.4	75.3	9.1	14.2	45.6	56.2
+ GRPO	71.5	86.4	68.8	81.9	10.4	16.4	50.2	61.6
+ OPD	73.1	85.2	69.3	79.4	13.5	16.8	52.0	60.5
+ StepCoder	74.0	86.0	70.3	80.1	14.2	17.2	52.8	61.1
+ PRLCoder	74.3	87.0	71.1	80.5	15.0	**20.6**	53.5	62.7
+ **MupCoder (Ours)**	**77.4**	**87.6**	**73.2**	**83.4**	**16.7**	20.3	**55.8**	**63.8**
Qwen2.5-Coder-3B								
Instruct	76.9	85.9	58.6	81.6	9.4	15.7	48.3	61.1
+ SFT	79.2	87.0	64.9	83.1	15.1	20.6	53.1	63.6
+ GRPO	82.3	90.9	70.2	86.3	15.5	23.2	56.0	66.8
+ OPD	83.2	88.4	70.8	85.7	17.8	24.8	57.3	66.3
+ StepCoder	84.5	89.1	71.4	85.9	17.4	24.1	57.8	66.4
+ PRLCoder	85.6	**91.7**	73.1	87.6	20.2	25.3	59.6	68.2
+ **MupCoder (Ours)**	**87.7**	**91.7**	**75.4**	**88.2**	**22.3**	**26.5**	**61.8**	**68.8**
Qwen2.5-Coder-7B								
Instruct	81.3	90.7	68.4	85.4	16.4	20.8	55.4	65.6
+ SFT	84.8	91.2	74.3	86.1	19.4	27.1	59.5	68.1
+ GRPO	87.4	93.5	76.6	90.1	22.1	32.2	62.0	71.9
+ OPD	89.1	93.7	77.2	90.9	24.8	31.9	63.7	72.2
+ StepCoder	89.4	94.2	77.0	88.4	24.4	31.5	63.6	71.4
+ PRLCoder	90.1	94.6	79.8	**92.5**	25.3	33.8	65.1	73.6
+ **MupCoder (Ours)**	**91.4**	**95.2**	**82.1**	91.7	**27.7**	**35.2**	**67.1**	**74.0**
Closed-Source References								
Gemini-2.5	90.1	96.2	80.6	94.5	64.8	72.9	78.5	87.9
GPT-5	91.3	97.1	80.6	94.7	66.0	73.4	79.3	88.4

Baselines. We compare MupCoder against a comprehensive suite of baselines spanning SFT, RL, OPD, and proprietary models. SFT (Ouyang et al., 2022) trains on the identical curated corpus using standard next-token prediction. GRPO (Shao et al., 2024) is implemented with sparse, outcome-only rewards derived from test-case execution, using a group size of 8. OPD (Lu & Lab, 2025) performs on-policy distillation using a frozen teacher initialized from Qwen2.5-Coder-32B-Instruct, providing token-level dense supervision without execution feedback. PRLCoder (Ye et al., 2025) provides line-level feedback through an auxiliary process reward model, while StepCoder (Dou et al., 2024) combines a code-completion curriculum with execution-aware masking for fine-grained policy optimization, with reproduction details provided in App. H. Furthermore, we compare our methods with OPSD (Zhao et al., 2026) in Sec. 3.3.

Implementation Details. All experiments are conducted on a cluster of 8 NVIDIA H100 (80GB) GPUs. We use the AdamW optimizer with a peak learning rate of 10^{-6} and a cosine learning rate decay schedule. The total global batch size is set to 256 and the training epoch is set to 2. For the surrogate objective, the clipping hyperparameter ϵ is set to 0.2. In the future-aware reward mechanism, we set the discount factor γ to 0.99 and the balancing coefficient λ to 1.0. During the rollout sampling phase, the temperature is maintained at 0.7. The prompt template is shown in App. K. All results are reported using the standard `pass@1` and `pass@10` metrics. We instantiate MupCoder on the Qwen2.5-Coder-Instruct model family at 1.5B, 3B, and 7B parameters.

3.2 Main Results

Tab. 1 compares MupCoder with representative post-training baselines across the Qwen2.5-Coder model family. Overall, the results suggest that MupCoder provides consistent gains in both generation accuracy and sampling-based performance.

MupCoder achieves the highest average Pass@1 across all three model scales. On the 7B model, it obtains an average Pass@1 of 67.1%, compared with 59.5% for SFT and 62.0% for outcome-only GRPO. This improvement indicates that augmenting sparse execution feedback with token-level dense signals can provide more informative optimization guidance. MupCoder also outperforms recent process-supervision baselines such as StepCoder and PRLCoder, improving over PRLCoder by 2.0 percentage points in average Pass@1 on the 7B model. Since these baselines typically operate at the line or segment level and often rely on auxiliary reward models, the results suggest that token-level self-distilled rewards, together with future-aware accumulation, can offer a useful form of fine-grained credit assignment for code generation.

Pass@10 provides an initial indication of sampling diversity, as it reflects the model's ability to produce a correct solution across multiple sampled generations. MupCoder achieves an average Pass@10 of 74.0% on the 7B model, compared with 68.1% for SFT and 71.9% for outcome-only GRPO. To measure diversity more directly, we further evaluate normalized-AST diversity, pairwise token-edit diversity, and the structural diversity of correct solutions. MupCoder performs best across all three metrics (App. G), showing that its improvement is accompanied by more diverse solution structures rather than being limited to the most likely output.

3.3 Ablation Study

To systematically evaluate the individual contributions of MupCoder's core components, we conduct fine-grained ablation studies on the Qwen2.5-Coder-7B model. We report the average Pass@1 and Pass@10 across HumanEval+, MBPP+, and LiveCodeBench-V6.

Multi-Perspective Hints. We evaluate the contribution of each teacher context by training MupCoder with one hint type at a time (Tab. 2). Each hint provides effective supervision, while combining all three achieves the best Pass@1 (67.1%) and Pass@10 (74.0%), indicating that demonstrations, pseudocode, and test cases provide complementary information. MupCoder also remains effective when solution-derived hints are removed or when test cases and retrieved demonstrations are degraded, with detailed results provided in App. E.

Table 2: Ablation studies on Qwen2.5-Coder-7B. Scores are averaged across the three benchmarks.

Configuration	Pass@1	Pass@10
MupCoder (Ours)	**67.1**	**74.0**
• *Multi-Perspective Hints*		
Only Tests (h_{test})	66.4	71.2
Only Demos (h_{demo})	65.4	71.8
Only Pseudocode (h_{pseudo})	65.9	70.6
Only Ground-truth Code	65.6	69.8
• *Future-Aware Reward Shaping*		
$\gamma = 0.90$ (Short-horizon)	66.2	73.4
$\gamma = 0$ (Myopic)	65.0	70.2
• *Hybrid Rewards*		
Only Dense ($\lambda = 0$)	64.2	73.5
Only Sparse	60.2	69.7

Future-Aware Reward Shaping. To evaluate the necessity of the future-aware mechanism, we ablate the discount factor γ to validate our future-aware return (Tab. 2). A strictly myopic setting ($\gamma = 0$) degrades Pass@1 to 65.0% and empirically causes response lengths to escalate, indicating insufficient penalty granularity for tokens that induce verbosity or nonsensical loops. While a short-horizon setting ($\gamma = 0.90$) partially recovers performance, it remains inferior to the full model ($\gamma = 0.99$). This indicates future-aware accumulation is crucial to penalize cascading errors and verbosity. Further hyperparameter experiments are detailed in App. B.

Hybrid Rewards. Finally, dissecting the hybrid reward (Tab. 2) reveals that sparse-only feedback yields the lowest performance (60.2% Pass@1) due to severe reward sparsity. Conversely, Dense Only rewards ($\lambda = 0$) maintain high diversity (73.5% Pass@10) but struggle with strict functional correctness (64.2% Pass@1), as they anchor to soft teacher distributions rather than absolute compiler truth. The hybrid integration cohesively bridges this gap: dense signals provide trajectory guidance, while sparse execution feedback anchors logical validity, culminating in consistent improvement in accuracy. Further hyperparameter experiments are detailed in App. B.

Comparison with On-Policy Self-Distillation. To examine the contribution of hinted teacher contexts, we compare MupCoder with OPD and OPSD variants using different hints. Unlike OPD, which relies on an external larger teacher, OPSD uses the same model conditioned on demonstrations, pseudocode, or test cases as its teacher. As shown in App. J, each single-hint OPSD variant performs worse than OPD, while combining all three hint types yields a modest improvement over OPD. This suggests that an individual hint provides useful but limited guidance, whereas combining complementary contexts strengthens self-teaching. The comparison also helps distinguish the benefits of hint-conditioned self-distillation from those of MupCoder's future-aware returns and execution-based rewards. Detailed results are provided in App. J.

3.4 Discussion

Training Efficiency. Beyond accuracy, MupCoder offers a favorable compute–performance trade-off. As shown in Fig. 4, our token-level dense reward provides more informative supervision during early optimization, thereby substantially accelerating convergence compared to sparse-reward baselines. Although estimating the multi-perspective reward requires three additional forward passes, this cost is offset by reducing more expensive sources of overhead. Standard GRPO samples 8 rollouts per prompt, incurring high generation and execution costs. OPD requires maintaining and querying an external 32B teacher model, while process-supervision methods such as PRLCoder require training a dedicated Process Reward Model (PRM) before RL. In contrast, MupCoder performs self-teaching with shared parameters, uses only a single rollout per prompt, and avoids PRM training. Therefore, it achieves faster convergence and strong final performance with a more economical training pipeline. Detailed rollout, execution, and runtime measurements are provided in App. C.

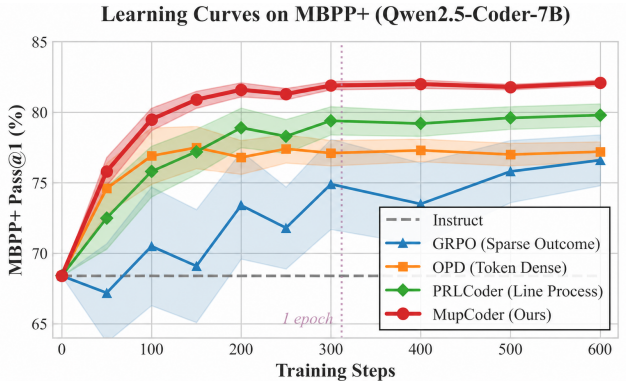

Figure 4: Learning curves and convergence dynamics of MupCoder compared to baselines. The solid lines indicate the mean performance, while the shaded regions represent the standard deviation across 5 independent runs.

Cross-Lingual Generalization. To examine whether the improvements transfer beyond Python, we evaluate Qwen2.5-Coder-7B on the C++ and Java splits of MultiPL-E (Cassano et al., 2022). All post-training methods use the same augmented Python corpus, without any C++ or Java post-training data. We conduct zero-shot evaluation and report Pass@1 at a sampling temperature of $T = 0.2$. As shown in Tab. 3, MupCoder achieves the best performance on both languages, reaching 81.2% on C++ and 80.5% on Java. It outperforms PRLCoder by 2.7 and 2.4 percentage points, respectively, suggesting that the capabilities acquired during Python post-training transfer to other programming languages.

Table 3: Zero-shot cross-lingual Pass@1 (%) on MultiPL-E using Qwen2.5-Coder-7B.

Method	C++	Java
Qwen2.5-Coder-7B Instruct	76.1	75.8
+ SFT	76.9	76.4
+ GRPO	77.8	76.9
+ OPD	78.0	77.1
+ StepCoder	78.3	77.4
+ PRLCoder	78.5	78.1
+ MupCoder (Ours)	**81.2**	**80.5**

Generalization to Reasoning Models. We further evaluate MupCoder on DeepSeek-R1-Distill-Qwen-7B to examine its generalization to reasoning models. MupCoder naturally supports reasoning traces because each hint-conditioned self-teacher scores the same trajectory sampled by the student, providing aligned token-level supervision over both reasoning and code tokens. As shown

in Tab. 4, MupCoder consistently outperforms outcome-only GRPO across all three benchmarks, improving the average Pass@1 from 66.7 to 71.0.

Table 4: Performance comparison on DeepSeek-R1-Distill-Qwen-7B. All methods use the same training data and evaluation protocol.

Method	HumanEval+	MBPP+	LCB-V6	Avg. Pass@1
DeepSeek-R1-Distill-Qwen-7B	84.1	70.2	29.6	61.3
+ GRPO	89.3	78.0	32.8	66.7
+ MupCoder (Ours)	**93.5**	**84.4**	**35.2**	**71.0**

We additionally evaluate MupCoder's zero-shot generalization to repository-level agentic coding and its robustness to hint's quality. MupCoder achieves consistent improvements in both settings, with detailed experimental setups and results provided in App. F and App. E.

4 Related Work

LLM-Based Code Generation. The application of LLMs to code generation has advanced rapidly, from early encoder-decoder architectures (Feng et al., 2020; Ahmad et al., 2021) to modern decoder-only models such as Codex (Chen et al., 2021), StarCoder (Li et al., 2023), and Qwen2.5-Coder (Hui et al., 2024). These models are typically trained via large-scale pre-training followed by SFT on curated instruction-following data (Wei et al., 2021; Chung et al., 2024). While SFT effectively aligns models with human coding preferences, it suffers from exposure bias (Bengio et al., 2015) and limited generalization to out-of-distribution programming tasks. To address these limitations, subsequent work has incorporated RL into code generation pipelines, employing outcome-based rewards derived from compilation or test-case execution (Le et al., 2022). However, such sparse binary feedback provides minimal guidance during the generation process: a single erroneous token can trigger cascading compilation or logical failures, yet the model does not receive intermediate supervision to locate or correct the mistake (Lu et al., 2025). This limitation has spurred interest in process supervision for denser intermediate guidance.

Process Supervision in Code Generation. The strict syntactic dependencies of programming languages have motivated a shift from coarse outcome rewards toward finer-grained process supervision. A prominent line of work trains Process Reward Models (PRMs) to score intermediate generation steps. For example, PRLCoder (Ye et al., 2025) automatically constructs line-level process-supervised data via statement mutation and refactoring, then trains a PRM to deliver per-line rewards during RL, significantly outperforming outcome-only baselines on MBPP and HumanEval. Similarly, CodePRM (Li et al., 2025) incorporates execution feedback into a PRM to score reasoning traces while StepCoder (Dou et al., 2024) decomposes long programs into completion subtasks with compiler-guided fine-grained optimization, and ORPS (Yu et al., 2024) unifies process and outcome rewards through tree search. Despite these advances, most existing approaches rely on auxiliary reward models and provide supervision at the line or segment level. In contrast, our work derives token-level dense signals directly from hint-conditioned self-distillation, avoiding separate PRM training mechanism.

5 Conclusion

We introduce MupCoder, a token-level reinforcement learning framework that mitigates reward sparsity in code generation through multi-perspective on-policy self-distillation. By conditioning the same model on complementary hints, MupCoder derives dense token-level supervision without a separate teacher or reward model. It combines discounted future alignment signals with execution-based rewards to support fine-grained policy optimization. Experiments across multiple model scales and code generation benchmarks demonstrate improved accuracy and faster convergence, with additional results indicating transfer to reasoning models, other programming languages, and repository-level coding tasks. These findings highlight the potential of combining contextual self-distillation with verifiable feedback for effective code-generation post-training.

AI USE STATEMENT

We used generative AI tools, including OpenAI Codex, to assist with drafting, revising, and organizing parts of the manuscript, including language polishing and paraphrasing. The tools were not used to generate experimental data or determine the reported scientific conclusions. All AI-assisted content was reviewed and revised by the authors, who verified the technical claims, numerical results, and citations against the underlying experiments and source materials.

REFERENCES

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 conference of the North American chapter of the association for computational linguistics: human language technologies*, pp. 2655–2668, 2021.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. *Advances in neural information processing systems*, 28, 2015.

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

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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, 25(70):1–53, 2024.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, et al. Stepcoder: improving code generation with reinforcement learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4571–4585, 2024.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

Hiroyuki Fudaba, Yusuke Oda, Koichi Akabe, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Pseudogen: A tool to automatically generate pseudo-code from source code. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 824–829. IEEE, 2015.

Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R Lyu. What makes good in-context demonstrations for code intelligence tasks with llms? In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 761–773. IEEE, 2023.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Joel Jang, Dongkeun Yoon, Sohee Yang, Sungmin Cha, Moontae Lee, Lajanugen Logeswaran, and Minjoon Seo. Knowledge unlearning for mitigating privacy risks in language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14389–14408, 2023.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35 (2):1–72, 2026.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. Codeprm: Execution feedback-enhanced process reward model for code generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 8169–8182, 2025.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Yaxuan Li, Yuxin Zuo, Bingxiang He, Jinqian Zhang, Chaojun Xiao, Cheng Qian, Tianyu Yu, Huanang Gao, Wenkai Yang, Zhiyuan Liu, et al. Rethinking on-policy distillation of large language models: Phenomenology, mechanism, and recipe. *arXiv preprint arXiv:2604.13016*, 2026.

Kevin Lu and Thinking Machines Lab. On-policy distillation. *Thinking Machines Lab: Connectionism*, 2025. doi: 10.64434/tml.20251026. https://thinkingmachines.ai/blog/on-policy-distillation.

Tingwei Lu, Yangning Li, Liyuan Wang, Binghuai Lin, Jiwei Tang, Wanshi Xu, Hai-Tao Zheng, Yinghui Li, Bingxu An, Zhao Wei, et al. From token to line: Enhancing code generation with a long-term perspective. *arXiv e-prints*, pp. arXiv–2504, 2025.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35: 27730–27744, 2022.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.

Chenxu Yang, Chuanyu Qin, Qingyi Si, Minghui Chen, Naibin Gu, Dingyu Yao, Zheng Lin, Weiping Wang, Jiaqi Wang, and Nan Duan. Self-distilled rlvr. *arXiv preprint arXiv:2604.03128*, 2026.

Yufan Ye, Ting Zhang, Wenbin Jiang, and Hua Huang. Process-supervised reinforcement learning for code generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 14224–14237, 2025.

Zhuohao Yu, Weizheng Gu, Yidong Wang, Zhengran Zeng, Jindong Wang, Wei Ye, and Shikun Zhang. Outcome-refining process supervision for code generation. *arXiv e-prints*, pp. arXiv–2412, 2024.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. Acecoder: Acing coder rl via automated test-case synthesis. *ArXiv*, abs/2502.01718, 2025.

Longhui Zhang, Bin Wang, Jiahao Wang, Xiaofeng Zhao, Min Zhang, Hao Yang, Meishan Zhang, Yu Li, Jing Li, Jun Yu, and Min Zhang. Function-to-style guidance of llms for code translation, 2025. URL https://arxiv.org/abs/2507.11083.

Longhui Zhang, Jiahao Wang, Chenhao Hu, Bingyu Liang, Jing Li, and Min Zhang. Bridging functional correctness and runtime efficiency gaps in llm-based code translation, 2026. URL https://arxiv.org/abs/2606.17683.

Siyan Zhao, Zhihui Xie, Mengchen Liu, Jing Huang, Guan Pang, Feiyu Chen, and Aditya Grover. Self-distilled reasoner: On-policy self-distillation for large language models. *arXiv preprint arXiv:2601.18734*, 2026.

Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, pp. 21–29, 2022.

A LIMITATION

The full MupCoder configuration leverages informative training-time contexts, including retrieved demonstrations, solution-derived pseudocode, and executable tests, whose availability and quality may vary across application settings. Nevertheless, our robustness experiments show that MupCoder degrades gracefully when solution-derived hints are removed or when test cases and retrieved demonstrations are incomplete, noisy, or less relevant (App. E). Like other execution-based reinforcement learning methods such as GRPO, MupCoder still requires a reliable verifier to provide outcome feedback, which limits its direct application to open-ended tasks without verifiable objectives. While our reasoning-model, cross-lingual, and agentic-coding results provide initial evidence of broader transfer, extending MupCoder to additional domains and alternative forms of hint remains an important direction for future work.

B HYPERPARAMETER SENSITIVITY ANALYSIS

To provide deeper insights into the stability and robustness of the MupCoder framework, we conduct a fine-grained sensitivity analysis on two critical hyperparameters: the balancing coefficient λ (which dictates the trade-off between the dense process supervision and the sparse outcome reward) and the discount factor γ (which controls the horizon of the future-aware reward mechanism).

Tab.5 presents the average Pass@1 and Pass@10 across our benchmark suite (HumanEval+, MBPP+, and LiveCodeBench-V6) for various configurations on the Qwen2.5-Coder-7B model.

For the balancing coefficient λ, the results indicate that the framework is relatively robust across a moderate range, peaking at our default setting of $\lambda = 1.0$. Lower values (*e.g.,* $\lambda \leq 0.5$) overly dilute the execution feedback, causing a slight drop in strict functional correctness (Pass@1) as the model leans too heavily on the soft teacher distributions. Conversely, excessively high values (*e.g.,* $\lambda = 3.0$) cause the sparse outcome reward to dominate the learning objective, which inadvertently reintroduces the optimization variance associated with reward sparsity, deteriorating both accuracy and diversity.

Regarding the discount factor γ, the performance demonstrates a high sensitivity to values near 1.0. While our primary ablation study (Sec. 3.3) established the necessity of long-term credit assignment over myopic settings, this fine-grained analysis reveals that a slightly decayed horizon ($\gamma = 0.99$) is optimal. A pure, undiscounted accumulation of future rewards ($\gamma = 1.00$) slightly degrades performance (Pass@1 drops to 66.5%).

In summary, maintaining an equilibrium between multi-perspective guidance and strict outcome validation ($\lambda \approx 1.0$), combined with a near-complete but gently decayed future awareness ($\gamma \approx 0.99$), is essential for maximizing the efficacy of MupCoder.

Table 5: Sensitivity analysis of the balancing coefficient (λ) and the discount factor (γ) on Qwen2.5-Coder-7B. The default configuration used in our main experiments ($\lambda = 1.0, \gamma = 0.99$) is bolded.

Configuration	**Avg. Pass@1**	**Avg. Pass@10**
Balancing Coefficient (λ) with $\gamma = 0.99$		
$\lambda = 0.1$	64.5	73.8
$\lambda = 0.5$	66.4	74.2
$\lambda = \mathbf{1.0}$ **(Default)**	**67.1**	**74.0**
$\lambda = 2.0$	65.8	73.1
$\lambda = 3.0$	63.5	72.2
Discount Factor (γ) with $\lambda = 1.0$		
$\gamma = 0.50$	64.1	71.5
$\gamma = 0.90$	66.2	73.4
$\gamma = 0.95$	66.7	73.8
$\gamma = \mathbf{0.99}$ **(Default)**	**67.1**	**74.0**
$\gamma = 1.00$	66.5	73.5

C COMPUTATIONAL COST

We directly compare the computational cost of MupCoder with all training baselines. Each method post-trains Qwen2.5-Coder-7B for two epochs on eight H100 80GB GPUs. We report the number of policy rollouts, online executions, and auxiliary scoring operations required for each training prompt, together with the total wall-clock time and GPU-hours. GPU-hours are calculated from the wall-clock time and the number of GPUs, providing a hardware-aware measure of total training cost.

Table 6: Direct training-cost comparison on Qwen2.5-Coder-7B. Executions are counted per training prompt.

Method	Rollouts	Executions	Auxiliary Scoring	Wall-Clock (h)	GPU-Hours
GRPO	8	8	None	31.1	248.8
OPD	1	0	$1\times$ 32B teacher	29.4	235.2
StepCoder	16	16	$16\times$ compiler feedback	45.8	366.4
PRLCoder-PPO	4	0	$4\times$ PRM scoring	41.6	332.8
MupCoder	**1**	**1**	$3\times$ **same 7B policy**	**22.4**	**179.2**

As shown in Tab. 6, the main computational overhead differs substantially across methods. GRPO relies on eight policy rollouts and executes every generated program, while StepCoder further increases this cost to 16 compiler-guided rollouts and executions. OPD reduces the number of rollouts but requires an auxiliary scoring pass through a larger 32B teacher. PRLCoder-PPO similarly introduces four scoring passes through a separately trained PRM.

MupCoder instead uses one policy rollout and one execution per prompt. Its three hint-conditioned scoring passes are performed through the same 7B policy used for rollout, without loading an external teacher or a separate reward model. Consequently, MupCoder obtains the lowest wall-clock time and GPU-hour cost among the compared methods. These results show that the additional hint-conditioned scoring is offset by the reduction in generation, execution, and auxiliary-model inference costs.

D HINT QUALITY ANALYSIS

As acknowledged in App. A, the efficacy of the MupCoder framework is intrinsically tied to the quality and relevance of the constructed multi-perspective hints. Because the student model derives its dense token-level supervision from the hint-conditioned teacher, noisy, misaligned, or overly explicit hints could severely skew the teacher's distribution and misguide the policy optimization. To rigorously assess the robustness of our data construction pipeline, we conduct a qualitative and quantitative analysis of the two primary generative hints: reference demonstrations (h_{demo}) and pseudocode (h_{pseudo}).

D.1 DEMONSTRATION EXTRACTION VIA BM25

To construct the demonstration hints (h_{demo}), we aim to provide the teacher with a worked example that shares an algorithmic reasoning pattern with the target problem, without leaking the direct solution. We utilized the Okapi BM25 retrieval algorithm to search our curated training corpus.

Implementation Details: We applied standard text preprocessing to the problem descriptions, including lowercasing, stop-word removal, and Snowball stemming. The BM25 parameters were empirically set to $k_1 = 1.5$ and $b = 0.75$ to balance term frequency and document length normalization. Crucially, the retrieval space was strictly limited to the training set ($\mathcal{D}_{aug}$), and the target problem itself was masked during the search to prevent exact-match leakage.

Quality Evaluation: To validate the semantic relevance of the retrieved demonstrations, we randomly sampled 300 target-retrieval pairs and had human software engineers manually evaluate them. The analysis revealed that in **84%** of the cases, the retrieved demonstration shared core algorithmic logic, data structures, or structural design patterns with the target problem.

Below is a representative example, illustrating how the retrieved task provides a relevant dynamic programming template without solving the target problem:

Example: BM25 Demonstration Retrieval (h_{demo})

Current Target Task (x):
Write a Python function to generate all valid combinations of n pairs of parentheses.

Retrieved Demonstration Task (h_{demo}):
Write a Python function to return all possible permutations of an array of distinct integers.

Retrieved Demonstration Code (Providing a Backtracking Template):

```
def permute(nums):
    result = []
    def backtrack(path, used):
        # Base case: valid combination found
        if len(path) == len(nums):
            result.append(path[:])
            return

        # Explore all possible choices
        for i in range(len(nums)):
            if used[i]:
                continue
            # 1. Make a choice
            used[i] = True
            path.append(nums[i])
            # 2. Backtrack (Recursive step)
            backtrack(path, used)
            # 3. Undo the choice
            path.pop()
            used[i] = False

    backtrack([], [False] * len(nums))
    return result
```

Impact of Retrieval Strategy: To further isolate the contribution of our BM25 retrieval mechanism and demonstrate the necessity of semantic relevance in hints, we conducted a controlled baseline experiment on the Qwen2.5-Coder-7B model. In this ablation, we replaced the BM25 retrieval with a *Random Selection* strategy, where the demonstration hint (h_{demo}) is sampled uniformly at random from the augmented training corpus $\mathcal{D}_{aug}$. This model was trained using only the demonstration hint, mirroring the setup in Sec. 3.3.

As shown in Tab. 7, substituting semantically relevant demonstrations with randomly sampled ones leads to a degradation in performance. The random contextual hints act as noise, forcing the teacher to condition on irrelevant algorithmic logic, which subsequently injects conflicting and misaligned token-level gradients into the student. The BM25 strategy outperforms the random baseline by a margin (+2.6% in Pass@1), confirming that the efficacy of demonstration hints inherently relies on retrieving structurally and algorithmically aligned reference paths.

Table 7: Comparison of demonstration retrieval strategies on Qwen2.5-Coder-7B (trained solely with h_{demo}).

Retrieval Strategy	Avg. Pass@1 (%)	Avg. Pass@10 (%)
BM25 Retrieval (Default)	**65.4**	**71.8**
Random Selection	62.8	69.5

D.2 Pseudocode Construction

The objective of the pseudocode hint (h_{pseudo}) is to distill the core algorithmic flow of the ground-truth solution while stripping away language-specific syntax and boilerplate (e.g., Python-specific APIs). To achieve this, we utilized the open-source `pseudogen` framework.

Mechanism: `pseudogen` approaches code-to-pseudocode conversion as a Statistical Machine Translation (SMT) problem. Rather than relying purely on brittle rule-based heuristics, it parses the Python source code into an Abstract Syntax Tree (AST) and uses a phrase-based machine translation model (built on top of the Moses decoder) trained on parallel code-to-description corpora. This allows the tool to map complex Python expressions into natural language algorithmic steps effectively, maintaining the causal control flow while masking the exact syntax required for execution.

Below is an example demonstrating how `pseudogen` transforms a standard Python snippet into an abstract algorithmic hint:

Example: Pseudocode Construction (h_{pseudo})

Ground-Truth Solution (Hidden from Teacher):

```
def get_gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

Generated Pseudocode Hint (h_{pseudo}):

```
define a function `get_gcd` with arguments `a` and `b`
while `b` is not equal to 0:
    simultaneously assign `b` to `a`, and the remainder of `a`
        divided by `b` to `b`
return `a`
```

E Robustness to Hint Quality and Availability

MupCoder uses demonstrations, pseudocode, and test cases as complementary training-time hints. We evaluate its robustness when these hints are unavailable, incomplete, noisy, or weakly related to the target problem.

E.1 Removing Solution-Derived Hints

The full configuration derives pseudocode from the target reference solution. To evaluate whether this supervision is essential, we remove the pseudocode hint or retain only test-case hints. The demonstration-plus-test variant does not use any hint derived from the ground truth solution, while the test-only variant additionally removes retrieved demonstrations.

Table 8: Performance without target-solution-derived hints.

Hint Configuration	Uses Target Solution	Avg. Pass@1	Avg. Pass@10
All Hints	Yes	**67.1**	**74.0**
Demonstrations + Tests	No	66.7	73.3
Test Cases Only	No	66.4	71.2

As shown in Tab. 8, removing target-solution-derived pseudocode causes only a small performance decrease. The test-only configuration also retains most of the pass@1 improvement, showing that MupCoder remains effective when reference solutions are unavailable.

E.2 Incomplete and Noisy Test Cases

We next evaluate robustness to imperfect test-case hints. For incomplete tests, we retain 100%, 80%, 50%, or 20% of the assertions for each task, with at least one assertion retained. The retained subset is used for both the test-case hint and execution reward, while evaluation always uses the complete suite.

For noisy hints, we randomly corrupt expected outputs while preserving their types. In this setting, corrupted tests are used only as hints, while the correct tests are retained for execution-based reward computation.

Table 9: Robustness to incomplete and noisy test-case hints.

Test-Case Condition	Avg. Pass@1
Incomplete Test Suites	
100% of assertions	**67.1**
80% of assertions	66.8
50% of assertions	65.9
20% of assertions	65.1
Noisy Test-Case Hints	
0% corrupted assertions	**67.1**
20% corrupted assertions	66.5

Tab. 9 shows that performance decreases gradually as test coverage is reduced. MupCoder also remains stable when 20% of the test-case hints are corrupted, indicating that moderate noise in a single hint channel does not destabilize training.

E.3 Demonstration Retrieval Quality

To assess sensitivity to retrieval quality, we replace BM25 retrieval with uniformly random demonstrations while keeping the demonstration-only training setting unchanged. We also remove training instances whose retrieved demonstrations have either problem-text embedding similarity or solution CodeBLEU above 0.8 and repeat the full MupCoder training.

Table 10: Robustness to retrieval quality and highly similar demonstrations.

Retrieval Condition	Avg. Pass@1	Avg. Pass@10
Demonstration-Only Training		
BM25 Retrieval	**65.4**	**71.8**
Random Retrieval	62.8	69.5
Full MupCoder Training		
Full Training Set	**67.1**	**74.0**
Remove Highly Similar Demonstrations	66.9	**74.0**

Semantically relevant retrieval provides stronger supervision than random demonstrations, as shown in Tab. 10. However, removing highly similar demonstrations has little effect on the full model, indicating that the gains are not driven by near-duplicate retrieved examples. Overall, MupCoder degrades gracefully when individual hint sources are removed or degraded rather than depending on any single informative context.

F Generalization to Agentic Coding

To examine whether the learned improvements transfer from single-turn code generation to repository-level software engineering, we evaluate all Qwen2.5-Coder-7B checkpoints on the full 500-task SWE-bench Verified benchmark (Jimenez et al., 2023). We use the same `mini-swe-agent` scaffold, system prompt, repository tools, interaction budget, decoding settings, patch-extraction procedure, and official evaluation harness for all methods. None of the checkpoints receives SWE-bench-specific or repository-level post-training, making this a zero-shot evaluation of transfer to agentic coding.

As shown in Tab. 11, MupCoder achieves the highest resolution rate, resolving 38 of 500 tasks. It outperforms the base checkpoint, outcome-only GRPO, and the compared process-supervision methods under the same agent scaffold. These results provide initial evidence that the capabilities

Table 11: Zero-shot evaluation on the full SWE-bench Verified benchmark using Qwen2.5-Coder-7B checkpoints and the same `mini-swe-agent` scaffold.

Method	Resolved Tasks	Resolution Rate
Qwen2.5-Coder-7B Instruct	24 / 500	4.8%
+ SFT	25 / 500	5.0%
+ GRPO	30 / 500	6.0%
+ OPD	31 / 500	6.2%
+ StepCoder	32 / 500	6.4%
+ PRLCoder	32 / 500	6.4%
+ MupCoder (Ours)	**38 / 500**	**7.6%**

learned through MupCoder can transfer to repository-level, tool-using coding tasks without agent-specific post-training.

G Solution Diversity Analysis

Pass@10 reflects whether a model can produce at least one correct solution from multiple samples, but it does not directly measure diversity among those samples. We therefore sample $K = 10$ programs per task and evaluate their structural and lexical diversity.

Let $y_{q,i}$ denote the i-th sampled program for task q, $A(y)$ its normalized abstract syntax tree (AST), and $T(y)$ its normalized token sequence. Before computing the metrics, we remove comments and formatting and canonicalize identifiers and literals while preserving operators and control flow. Invalid programs share a common `INVALID` AST signature.

We consider the following three metrics:

Normalized-AST Diversity. We measure the fraction of distinct normalized AST signatures among the K samples:

$$D_{\mathrm{AST}} = \frac{1}{|Q|} \sum_{q \in Q} \frac{\left|\{A(y_{q,i})\}_{i=1}^{K}\right|}{K}. \tag{9}$$

Token-Edit Diversity. We compute the mean pairwise normalized Levenshtein distance between token sequences:

$$D_{\mathrm{edit}} = \frac{1}{|Q|} \sum_{q \in Q} \frac{2}{K(K-1)} \sum_{i<j} \frac{\mathrm{Lev}(T(y_{q,i}), T(y_{q,j}))}{\max(|T(y_{q,i})|, |T(y_{q,j})|)}. \tag{10}$$

Correct-Solution Structural Diversity. Let C_q be the set of samples that pass the complete evaluation suite, $n_q = |C_q|$, and $g_q = |\{A(y) : y \in C_q\}|$. We measure the fraction of structurally distinct solutions among all correct samples:

$$D_{\mathrm{correct}} = \frac{\sum_q g_q}{\sum_q n_q}. \tag{11}$$

As shown in Tab. 12, MupCoder achieves the highest diversity under all three metrics. Its improvement in normalized-AST and token-edit diversity indicates greater variation in both program structure and token-level realization. MupCoder also obtains the highest correct-solution structural diversity, showing that the increase is retained when considering only functionally correct programs and is not merely a consequence of producing more correct samples.

Table 12: Solution diversity with $K = 10$ samples per task. Higher values indicate greater diversity.

Method	$\mathbf{D}_{\text{AST}}$	$\mathbf{D}_{\text{edit}}$	$\mathbf{D}_{\text{correct}}$
SFT	57.8%	0.34	27.6%
GRPO	62.5%	0.38	30.3%
PRLCoder	64.0%	0.40	31.3%
MupCoder (Ours)	**67.6%**	**0.44**	**34.4%**

H Baseline Reproduction Details

To guarantee a rigorous and equitable evaluation, we meticulously reproduced the state-of-the-art process supervision baselines—PRLCoder and StepCoder—adapting them to the Qwen2.5-Coder model family utilized in our primary experiments. Our reproduction protocols strictly adhered to the methodologies outlined in their original publications. By standardizing the foundation models and the training corpus, we reduce confounding factors related to model scale and data quality, making the observed gains more directly comparable across methods.

H.1 Reproduction of PRLCoder

PRLCoder leverages process-supervised reinforcement learning by utilizing an automated pipeline to generate line-level reward signals, addressing the high annotation cost of traditional process supervision.

Implementation Details: Following the methodology described by Ye et al. (2025), we implemented their "statement mutation/refactoring-compile and execution verification" strategy. We generated mutated and refactored code variations line-by-line for our augmented training corpus, utilizing compiler and execution feedback to automatically assign deterministic line-level correctness labels. Subsequently, we trained a dedicated Process Reward Model (PRM), initialized from the corresponding Qwen2.5-Coder Instruct checkpoints (1.5B, 3B, and 7B), using this annotated dataset. During the reinforcement learning phase, we deployed this PRM to provide intermediate line-level rewards to the policy model. This setup serves as the direct baseline representing line-level auxiliary reward models against MupCoder's token-level self-distilled signals.

H.2 Reproduction of StepCoder

StepCoder enhances RL exploration in long-sequence code generation by integrating compiler feedback with curriculum learning and execution-aware masking.

Implementation Details: Following Dou et al. (2024), we implemented the two core components: the Curriculum of Code Completion Subtasks (CCCS) and Fine-Grained Optimization (FGO). For CCCS, we constructed code-completion subtasks by providing partial reference-code prefixes and progressively reducing the provided context, transitioning from easier completion tasks toward full program generation. During policy optimization, we applied the FGO masking strategy to exclude unexecuted code segments from the policy loss. This restricts policy updates to tokens in segments executed during testing, including those in programs that ultimately fail the tests. We maintained our standard hyperparameter settings (*e.g.,* learning rate and batch sizes) while adapting rollout generation to the code-completion curriculum.

I Hint Informativeness

To assess whether the constructed hints provide useful task information, we conduct a diagnostic evaluation on a randomly sampled subset of 2,000 training problems. We evaluate Qwen2.5-Coder-7B with each hint type prepended to the problem prompt and compare its pass@1 with the no-hint setting. We also evaluate Qwen2.5-Coder-32B without hints as a reference.

As shown in Tab. 13, all three hint types improve the 7B model's pass@1, with pseudocode yielding the largest gain, followed by test cases and demonstrations. The pseudocode- and test-conditioned

Table 13: Hint informativeness on 2,000 randomly sampled training problems. Hints are supplied as additional input context for this diagnostic evaluation. Pass@1 is reported in percent.

Setting	**Pass@1 (%)**
Qwen2.5-Coder-7B (No Hint)	53.8
+ Demonstrations (h_{demo})	62.9
+ Pseudocode (h_{pseudo})	72.4
+ Test Cases (h_{test})	68.7
Qwen2.5-Coder-32B (No Hint)	64.9

Table 14: Comparison of OPD, OPSD with different hints, and MupCoder on Qwen2.5-Coder-7B. Results are averaged across HumanEval+, MBPP+, and LiveCodeBench-V6 and reported in percent.

Method	**Teacher Context**	**Avg. Pass@1**	**Avg. Pass@10**
OPD	No hints (external 32B teacher)	63.7	72.2
OPSD	Demonstrations only	62.8	71.4
OPSD	Pseudocode only	62.1	70.8
OPSD	Test cases only	63.2	72.2
OPSD	All three hints	63.7	72.8
MupCoder	All three hints	67.1	74.0

7B variants also outperform the 32B model without hints in this diagnostic setting. These results support the informativeness of the teacher-side contexts, but do not by themselves establish the reliability of token-level supervision on student-generated prefixes.

J Comparison with On-Policy Self-Distillation

We compare OPD, OPSD, and MupCoder using Qwen2.5-Coder-7B under the same training corpus and evaluation protocol as the main ablation study. OPD uses an external Qwen2.5-Coder-32B-Instruct teacher, whereas OPSD conditions the same backbone model on information to obtain teacher-side supervision.

For OPSD, we follow the sampled-token policy-gradient formulation presented in the original work, using reverse KL divergence between the student and teacher policies. Specifically, the immediate token-level distillation signal is the teacher–student log-probability difference evaluated at each student-sampled token. To examine the effect of teacher context, we use demonstrations, pseudocode, and test cases individually, as well as all three hint types. For the all-hint configuration, we average the three hint-conditioned log-probability differences as in Eq. (4). All OPSD configurations use these immediate signals without future accumulation or execution rewards, corresponding to $\gamma = 0$ and $\lambda = 0$ in our notation.

We report average pass@1 and pass@10 across HumanEval+, MBPP+, and LiveCodeBench-V6.

As shown in Tab. 14, combining all three hint types enables OPSD to slightly outperform OPD, while each single-hint variant falls below OPD. This suggests that complementary informative contexts can strengthen self-teaching beyond what an individual hint provides.

K Evaluation Prompt Templates

We use task-specific zero-shot prompts for single-turn code generation, cross-lingual code generation, reasoning-model evaluation, and repository-level issue resolution. Within each evaluation setting, the prompt template and output requirements are kept identical across the compared checkpoints. Evaluation prompts do not include the training-time demonstration, pseudocode, or additional test-case hints. Information provided in the original benchmark task is retained.

Python Code Generation. For HumanEval+, MBPP+, and LiveCodeBench-V6, the task prompt contains the benchmark-provided problem statement, together with any function signature, starter code, or input/output requirements.

Python Code Generation Prompt

User Prompt:

```
Please complete the following Python code to solve the problem.
Only output the valid Python code enclosed within a markdown code
block, without any conversational padding or additional
explanations.

{task_prompt}
```

Cross-Lingual Code Generation. For MultiPL-E, we use language-specific prompts for C++ and Java. The task prompt includes the benchmark-provided target-language problem description and code scaffold. The prompts instruct the model to preserve the interfaces required by the evaluation harness.

C++ Code Generation Prompt

User Prompt:

```
Complete the following C++ code to solve the problem.

Preserve the provided function signature and any required interface
. Include any necessary headers. Do not add a main function unless
the task requires one.

Only output the completed C++ solution enclosed within a markdown
code block, without additional explanations or test code.

{task_prompt}
```

Java Code Generation Prompt

User Prompt:

```
Complete the following Java code to solve the problem.

Preserve the provided class name, method signature, access
modifiers, and any required interface. Include any necessary
imports. Do not add a main method unless the task requires one.

Only output the completed Java solution enclosed within a markdown
code block, without additional explanations or test code.

{task_prompt}
```

Reasoning-Model Evaluation. For DeepSeek-R1-Distill-Qwen-7B, we allow reasoning before the final solution and request a single Python code block as the final answer. The execution harness evaluates the extracted program rather than the preceding reasoning text.

Reasoning-Model Code Generation Prompt

User Prompt:

```
Solve the following programming problem in Python.

You may reason through the problem before providing your final
solution. Follow the specified function signature or input/output
format.

End your response with the complete solution in a single markdown
code block labeled python. Keep any reasoning outside this code
block and do not include additional text after it.

{task_prompt}
```

Repository-Level Agentic Coding. For SWE-bench Verified, the model interacts with the repository through the mini-swe-agent scaffold. The task message supplies the issue description without reference patches or training-time hints. The scaffold provides the tool interface and action-format instructions, which are shared across all compared checkpoints. Unlike the single-turn settings above, the agent produces its solution by modifying repository files.

Repository-Level Issue Resolution Prompt

User Prompt:

```
Resolve the following issue in the repository available in your
working environment.

Inspect the relevant source files and tests, identify the cause of
the issue, and make the necessary changes directly in the
repository. Preserve existing behavior outside the scope of the
issue and follow the repository's conventions.

Use the available tools to run relevant tests when feasible. Do not
 modify existing tests merely to make a failing implementation pass
.

Follow the agent scaffold's action format and completion procedure.
 The solution will be evaluated from the changes to the repository
files.

Issue description:
{issue_description}
```