CoT-Edit: Reinforcement Learning of Chain-of-Thought Reasoning for Code Edit Suggestion

Wuya Chen[1*], **Yihao Yang**[1,2], **Yue Lin**[1]
[1]Netease Guangzhou AI Lab
[2]South China University of Technology
chenwuya@corp.netease.com

Abstract

Code edit suggestion, which encompasses modifying, refactoring, and maintaining existing code, represents the most frequent software development activity and has become a focal point for AI-powered tools. Traditional methods translate explicit natural language instructions into code edits, while pattern-based approaches learn from users' historical editing patterns to provide style-consistent and more accurate suggestions. However, these pattern-based methods still face two critical challenges: (1) difficulty handling edits that demand deep contextual reasoning, and (2) lack of interpretability in editing decisions. To tackle this, we propose CoT-Edit, a reinforcement learning framework that guides LLMs to discover chain-of-thought (CoT) reasoning paths for code editing without requiring human-annotated CoT data. Specifically, we design multi-step reasoning framework that enable: (1) analysis-guided code editing, and (2) seamless switching between CoT and non-CoT inference modes. Building on this, we introduce Edit-Aware Reward Modeling (EARM), a fine-grained diff-based reward approach for effective learning. Furthermore, we discover a LoRA merging strategy that enhances model generalization. Evaluations on an industrial dataset show that our approach achieves 60.2% edit accuracy, outperforming all strong baselines. Online A/B tests further confirm its effectiveness in production. Code is available at `https://github.com/202230483077yyh/CoT-Edit`.

1 Introduction

Code edit suggestion, which encompasses optimization, refactoring, and debugging, represents a substantial portion of developers' daily activities (Nguyen et al., 2013). Recent advances in Large Language Models (LLMs) have enabled AI-powered development tools such as GitHub

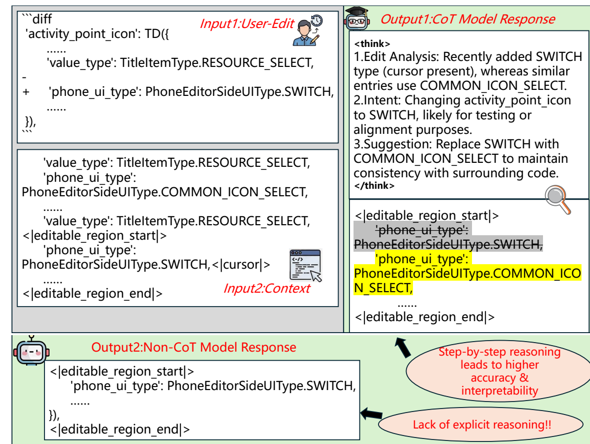

Figure 1: An example illustrating that explicit reasoning improves accuracy while providing inspectable reasoning traces.

Copilot (Copilot, 2023), Cursor (Cursor, 2024), Claude code (Anthropic), and Zeta (Zed, 2025) to transform code editing workflows. Current code edit models fall into two main categories: (1) **Instruction-based models** (Li et al., 2024), which translate explicit natural language instructions into code edits. However, these methods require continuous human guidance to maintain context coherence, thereby disrupting programming flow and increasing cognitive load. (2) **Pattern-based approaches** (Zed, 2025; Liu et al., 2024b; Li et al., 2024; Chen and et al., 2025), which learn from users' historical editing behaviors to provide style-consistent suggestions. Unlike instruction-based models that require explicit user guidance, pattern-based approaches offer a more seamless alternative by automatically generating context-aware suggestions from editing sequences, thereby reducing both latency and cognitive overhead while significantly improving development efficiency. This efficiency advantage is particularly critical as code editing tasks are inherently time-sensitive, with users expecting feedback within 1 second to maintain flow state (Card, 2018). Nevertheless, current

*Corresponding Author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 28219–28234
July 2-7, 2026 ©2026 Association for Computational Linguistics

pattern-based approaches remain constrained by two significant challenges: First, they exhibit **limited semantic comprehension**, struggling to handle complex edits that require understanding code semantics beyond surface-level patterns. Second, they suffer from **insufficient interpretability**, providing suggestions without explanatory rationale, which complicates user verification and reduces trust in automated recommendations.

In this work, we reformulate code editing as an explicit reasoning task, where edits are derived through structured logical inference rather than directly predicting edit suggestions. To realize this vision, we introduce a novel Reinforcement Learning (RL)-based training paradigm that grounds model decisions in interpretable reasoning steps, achieving significant improvements in editing accuracy (see Figure 1). To ensure models capture real-world developer behavior, we first construct a data curation pipeline that distills high-quality training examples from production user logs. Building on this foundation, we design a multi-step reasoning framework that elicits chain-of-thought (CoT) reasoning during training while enabling seamless switching between CoT and non-CoT modes during inference—critical for practical deployment scenarios with varying latency constraints. Tailored to the unique characteristics of code editing, we further propose an edit-aware reward function that guides RL optimization toward meaningful, localized modifications. Finally, we introduce a novel LoRA-merge mechanism that combines the strengths of code-chat and code-base models. We train LoRA adapters on code-chat to acquire reasoning capabilities via RL, then merge them into code-base to retain its superior completion performance. Extensive offline experiments on an industrial dataset, along with online A/B tests, demonstrate that our approach consistently outperforms strong baselines and establishes a practical design paradigm for RL-grounded reasoning in code editing.

Our key contributions are summarized as follows:

- To our knowledge, we are the first to reformulate code editing—as distinct from code generation—as an explicit reasoning task, introducing a novel RL-based training paradigm that elicits structured CoT outputs, improving accuracy while providing inspectable reasoning traces.
- We design a multi-step reasoning framework that elicits CoT reasoning during training and enables flexible CoT/non-CoT switching during inference for different latency needs.
- We propose an edit-aware reward function tailored to code editing characteristics, guiding RL optimization toward localized, meaningful modifications.
- We introduce a LoRA-merge mechanism that transfers reasoning capabilities from code-chat models to code-base models via RL-trained adapters, enabling CoT reasoning while retaining superior completion performance.
- We validate our approach through extensive offline experiments on an industrial dataset and online A/B tests, demonstrating consistent gains over strong baselines.

2 MOTIVATING EXAMPLE

```
<button onclick="concatenateContents() generateOutput()">        Edit 1
    Generate
</button>
</div>
<script>
    function concatenateContents() generateOutput() {           User-Edit Modification
      ......
    }
    // On start, generate output:
    concatenateContents() generateOutput();                      Edit 2

    function flipContents() {
      ......
      concatenateContents() generateOutput();                    Edit 3
    }
</script>
```

Figure 2: A motivating example of pattern-based code edit.

This section motivates pattern-based code edit suggestion through a representative refactoring task in modern front-end development. Consider a developer renaming the function identifier from `generateOutput()` to `concatenateContents()` to better reflect its string concatenation behavior (see Figure 2). The developer must perform the following laborious steps: (1) Initial Modification: Rename the function definition. (2) Edit 1: Update the function call in the <button> element. (3) Edit 2: Replace the function reference within the <script> tag. (4) Edit 3: Synchronize the reference in `flipContents()`. This process requires extensive file navigation and manual search-and-replace operations, while demanding cognitive effort to verify all references are correctly updated.

Pattern-based code edit suggestion addresses these issues by capturing the developer's initial edit to predict intent and automatically suggest sub-

sequent modifications without explicit instructions: (1) Edit Interception: Detect the rename operation after the initial edit. (2) Intent Inference: Identify the rename pattern and infer the intent to synchronize all references. (3) Edit Suggestion: Proactively suggest modifications when the cursor reaches relevant locations. (4) Seamless Continuation: Confirm suggestions via Tab key, iteratively completing all updates. This approach transforms the laborious manual workflow into a fluid interaction paradigm, reducing cognitive burden and enabling a coherent development experience.

3 METHODS

This section outlines our RL-based training paradigm (Figure 3). We first introduce the dataset construction pipeline (§3.1), followed by the model training process (§3.2).

3.1 Dataset Construction

To ensure our constructed data better reflects real-world usage scenarios, we developed an efficient data collection mechanism within our AI-powered IDE that accurately captures developers' code edit trajectories during actual programming activities. The mechanism consists of two steps: 1) Difference Modeling, 2) Derivation-Based Filtering.

3.1.1 Task Formulation

Our process aims to transform the continuous sequence of code changes into separate, ordered training examples. Here, we consider consecutive timestamp pairs, denoted as $T-1$ and T, wherein developer operations (such as edits) trigger state transformations. Every transformation produces a data tuple comprising: 1) The pre-edit code state (C_{T-1}), 2) The historical editing trajectory up to time $T-1$ (denoted as H_{T-1}), 3) The current edit E_T, i.e., the code modification performed between states C_{T-1} and C_T.

We formulate code edit suggestion model as π_θ, which takes the editing history H_{T-1} and pre-edit code C_{T-1} as input, and outputs the full updated code:

$$C_T = \pi_\theta(H_{T-1}, C_{T-1})$$

Crucially, C_T denotes the complete code state after edits, rather than incremental changes (diff).

3.1.2 Difference Modeling

To enable real-time responsiveness while maintaining contextual precision, similar to prior works (Zed, 2025; Chen and et al., 2025), we employ an incremental approach that computes differences only on actively modified code segments rather than entire files. **Difference Calculation**, our AI-powered IDE captures edit regions in real-time and applies a sequence-matching algorithm to these localized segments, enabling instant change detection. We utilize the standard diff format to provide models with unambiguous context for code modifications. **Difference Merge**, we merge overlapping differences using developer-edited code blocks as units, creating cohesive recommendations that align with real-world editing behaviors and avoid fragmented suggestions. The detailed algorithmic procedure for this merging process is provided in Appendix A.

3.1.3 Derivation-Based Filtering

A critical issue inherent in raw edit trajectory data is the presence of spurious correlations. Developer edit sequences often lack causal coherence: a developer may transition directly from completing one task to initiating an entirely unrelated task. Training models on such temporally adjacent yet semantically disconnected sequences risks inducing spurious pattern learning, potentially generating misleading and contextually irrelevant recommendations. To tackle this, we design a fine-grained data refinement pipeline. Specifically, we employ multiple LLMs as relevance filters through carefully designed prompt (see Figure 4 in Appendix C) to determine whether the current edit (E_T) is causally derivable from the **historical edits** (H_{T-1}) or the **current code state** (C_{T-1}). When all models consistently agree that the current edit represents a logical and predictable continuation, we classify E_T as a relevant edit and retain it in the training set. Otherwise, the instance is discarded as potentially spurious.

3.2 Model Training

To enable the model to generate inspectable reasoning paths and accurate edit predictions, we design a three-stage training pipeline. First, we introduce a multi-step reasoning framework that decomposes the edit suggestion task into explicit reasoning stages (§3.2.1). Second, we employ reinforcement learning (RL) to enhance the model's reasoning and generalization capabilities beyond supervised fine-tuning (SFT), using an edit-aware reward function tailored to code edit scenarios (§3.2.2). Finally, we propose a novel LoRA merging mechanism to

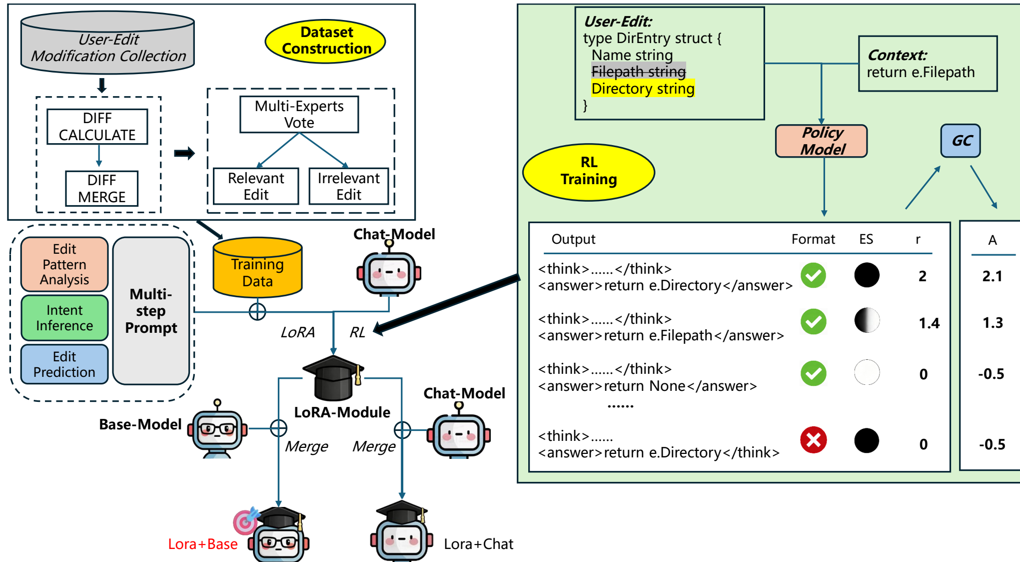

Figure 3: Overview of our RL-based training paradigm. GC denotes group computation in GRPO, while Format and ES represent format reward and correctness reward, respectively.

transfer reasoning capabilities from code-chat models to code-base models, preserving both reasoning quality and completion performance (§3.2.3).

3.2.1 Multi-step Framework Design

To enable the model to automatically generate explicit reasoning processes, we design a structured prompt (see Figure 5 Appendix D) that guides the model to progressively decompose the code edit suggestion task into three interpretable reasoning steps: (1) **Edit Pattern Analysis**: The model examines the edit history to identify the developer's coding patterns and assess the current code state. (2) **Intent Inference**: Based on the pattern analysis, the model infers the developer's underlying intent and what they are likely attempting to accomplish. (3) **Edit Prediction**: Leveraging insights from the previous two steps, the model predicts the most appropriate edit suggestion as the next logical step.

Furthermore, to enable seamless switching between CoT and non-CoT modes during deployment, we employ a dual-tag structure during training: reasoning steps are enclosed within `<think>` and `</think>` tags, while the final edit output is delimited by `<answer>` and `</answer>` tags. During inference, this architecture enables flexible mode control—CoT mode is activated by initiating generation with the `<think>` tag to elicit explicit reasoning, while non-CoT mode is triggered by directly starting with the `<answer>` tag, thereby bypassing intermediate reasoning steps.

3.2.2 RL Training

While the multi-step prompt enables structured reasoning, supervised fine-tuning alone faces two key challenges: (1) the scarcity of large-scale annotated CoT data, and (2) limited generalization to unseen edit scenarios. To address these limitations, we introduce RL with an edit-aware reward function that allows the model to refine its reasoning through trial-and-error.

Edit-Aware Reward Modeling Inspired by existing work (DeepSeek-AI and et al., 2025), we design our reward function with two components: format and correctness. Notably, the correctness reward is diff-based, focusing on edit-level similarity, where edits represent diff-type content operations, rather than overall code content similarity.

Format Reward As outlined in the previous section, our model outputs adhere to the following format:

`<think>...</think><answer>...</answer>`.

That is, the model is required to output its reasoning process within the `<think>` tag and the final solution within the `<answer>` tag. The format reward is defined as follows:

$$R_{\text{format}}(y) = \begin{cases} +1, & \text{if } y \text{ meets the format} \\ -1, & \text{otherwise} \end{cases} \quad (1)$$

where y is the output of the model.

Correctness Reward The correctness reward is designed to favor perfect accuracy while still rewarding semantically similar edits with partial credit. Furthermore, to precisely quantify edit quality, we employ diff-based similarity metrics rather than full code similarity. This distinction is critical: when measuring full code similarity, minimal edits (e.g., modifying a single line) can produce deceptively high similarity scores (>95%) because most code remains unchanged. In contrast, diff-based similarity focuses exclusively on the modified portions, accurately capturing what was actually changed in that single line (e.g., which specific API call was added). See Appendix E for a detailed comparison. Thus, correctness reward is defined as:

$$R_{\text{cor}}^{\text{rule}} = \begin{cases} 1.0 & \text{if } s = 1.0 \text{ (perfect match)} \\ \alpha \times s & \text{if } s > \beta \text{ (partial match)} \\ -1.0 & \text{if } s \leq \beta \text{ (low similarity)} \end{cases} \quad (2)$$

where $s = \text{ES}(E_{\text{pred}}, E_{\text{gt}})$ denotes the edit similarity score. The hyperparameters α and β control the reward scaling and similarity threshold.

Additionally, To complement the precision-oriented rule-based metric and mitigate potential false negatives in string matching, we introduce an LLM-based evaluator for cases where rule-based similarity is low ($s \leq \beta$). Specifically, we employ Qwen2.5-Coder-32B-Instruct (Hui and et al., 2024) as an external judge to assess semantic equivalence:

$$R_{\text{cor}}^{\text{LLM}} = \text{LLM}(E_{\text{pred}}, E_{\text{gt}}) \quad (3)$$

where the LLM judge produces a binary correctness score $R_{\text{cor}}^{\text{LLM}} \in \{0, 1\}$, with 1 indicating semantic equivalence and 0 otherwise. The final correctness reward is:

$$R_{\text{cor}} = \begin{cases} R_{\text{cor}}^{\text{rule}} & \text{if } s > \beta \\ R_{\text{cor}}^{\text{LLM}} & \text{if } s \leq \beta \end{cases} \quad (4)$$

However, empirical results show that LLM-based judgment does not improve performance in our setting. Consequently, our final system relies solely on the rule-based reward $R_{\text{cor}}^{\text{rule}}$, which provides both computational efficiency and sufficient accuracy for code edit scenarios.

Our reward is applicable to any RL algorithm that uses policy gradient updates. In our experiments, we select GRPO (Shao et al., 2024) as the training algorithm.

3.2.3 LoRA Merging

Code edit suggestion is an extension of code completion, and existing edit models are typically fine-tuned from code completion models. Meanwhile, code models generally fall into two categories: (1) code-base models, pre-trained specifically for completion tasks, and (2) code-chat models, instruction-tuned variants that exhibit stronger instruction-following capabilities but degraded completion performance. This presents a dilemma: training on code-chat benefits reasoning but sacrifices completion quality, while training on code-base preserves completion but limits reasoning capability.

To address this trade-off, we propose a novel LoRA merging mechanism. Specifically, during the RL phase, we train LoRA adapters on top of code-chat to leverage its instruction-following strength for multi-step reasoning. The learned LoRA weights are then merged into code-base, enabling the model to acquire CoT reasoning capabilities while largely preserving its original code completion performance. A detailed analysis of this merging mechanism is provided in §4.4.

4 Experiments

4.1 Experimental Settings

4.1.1 Data Preparation and Metrics

We aim to develop code edit models tailored for real-world development scenarios. To this end, we leverage our internally developed AI-powered IDE, which is actively used by over 5,000 developers daily, providing us with extensive user interaction logs. Following the data construction pipeline described in §3.1, we curate a high-quality dataset of 1,840 training samples and 670 test samples from these logs. Inspired by the finding of LIMR (Li et al., 2025) that RL training benefits more from data quality than quantity, we prioritize quality over scale: the training set is manually refined to ensure diversity across more than 10 edit categories and a balanced difficulty distribution.

To ensure test set quality and prevent data leakage, we implement three safeguards: (1) manual review to verify correctness and representativeness; (2) temporal separation between training and test samples; and (3) deduplication based on code context to eliminate near-duplicates across splits. We denote this curated test set as **Edit-Test** for offline evaluation.

Given that Python and Java account for over 70% of production traffic, we focus our efforts

exclusively on these two languages.

Offline We use Exact Match Rate (EMR) as the offline evaluation metric for the code edit task, defined as:

$$\text{EMR} = \frac{1}{N} \sum_{i=1}^{N} 1 \left[\hat{E}_i = E_i \right], \tag{5}$$

where $\hat{E}_i$ represents the predicted edit and E_i denotes the ground-truth edit for the i-th sample, and $1[\cdot]$ indicates exact string matching.

Online To evaluate the online performance of our model in real-world scenarios, we conduct A/B testing on our internally developed AI-powered IDE platform. We randomly allocated 25% of traffic to each of the following: Treatment Group 1 (our first approach), Treatment Group 2 (our second approach), and Control Group (a strong baseline model).

We adopt the general business metric Acceptance Rate (AR) as the online evaluation, which measures the proportion of model suggestions accepted by users:

$$\text{AR} = \frac{\sum_{i=1}^{M} A_i}{\sum_{i=1}^{M} R_i}, \tag{6}$$

where A_i and R_i are accepted and total suggestions for user i, and M is the number of users.

4.1.2 Models

We compare our approach against the following baselines. For brevity, we denote Qwen2.5-Coder-7B-Base and Qwen2.5-Coder-7B-Chat as **Qwen2.5-Base** and **Qwen2.5-Chat**, respectively, and Qwen3-Coder-480B-A35B-Instruct (Yang and et al., 2025) as **Qwen3-Coder-480B**.

- **Qwen3-Coder-480B**: MoE model (480B total, 35B active) developed by the Qwen team for code generation and tool use.
- **Claude** (Anthropic): Claude Sonnet 4, an AI assistant by Anthropic with robust coding capabilities.
- **Zeta** (Zed, 2025): Instruction-free edit generator leveraging user edit history (fine-tuned on Qwen2.5-Base).
- **SFT**: We finetune Qwen2.5-Base (Hui and et al., 2024) on 1840 (H_{T-1}, C_{T-1}, C_T) triples, directly predicting final edits without reasoning traces.
- **SFT-CoT**: We use Claude to synthesize CoT reasoning traces from input code and the ground-truth edit in the training set for supervised fine-tuning of Qwen2.5-Base (Hui and et al., 2024).
- **CoT-Edit-Merge.Base**: Our code edit model is developed through a two-stage process: first, we enhance Qwen2.5-Chat (Hui and et al., 2024) with CoT reasoning capabilities via reinforcement learning; second, we merge the resulting model into the Qwen2.5-Base backbone to produce the final system.
- **CoT-Edit-Merge.Chat**: Following the same two-stage approach as CoT-Edit-Merge.Base, this variant differs in the second stage by merging the resulting model into the Qwen2.5-Chat backbone instead of the Base backbone.
- **CoT-Edit-Merge.Code-RM**: Identical to CoT-Edit-Merge.Base except replacing our proposed Edit-Aware Reward Modeling with overall code content similarity.

Additional implementation details can be found in Appendix B.

4.2 Evaluation

4.2.1 Overall Performance

As shown in Table 1, our proposed RL-CoT method achieves the best overall performance across all evaluation metrics. Specifically, CoT-Edit-Merge.Base attains 60.2% average EMR with CoT inference, substantially outperforming both Claude (45.9%) and Qwen3-Coder-480B (40.5%) by 14.3 and 19.7 percentage points, respectively. Moreover, the performance remains strong (59.8%) even without CoT inference. Importantly, the offline performance gains are validated in online deployment. CoT-Edit-Merge.Base achieves 19.36% acceptance rate (Table 2), a 172% relative improvement over the baseline (7.1%), validating the real-world effectiveness of our approach.

RQ1: *Does our dataset better align with online user behavior?* Zeta and SFT are both trained using supervised fine-tuning with different training data: Zeta uses the dataset constructed by the Zed team, while SFT uses our dataset derived from online user logs, which better aligns with real-world usage scenarios. SFT significantly outperforms Zeta in both offline (51.1% vs. 40.6%, Table 1) and online (16.95% vs. 7.1%, Table 2) evaluations, achieving relative improvements of 25.7% and 138.7% respectively. This demonstrates that

Model	Python	Java	Average
Non-COT Models			
Zeta	44.3	37.0	40.6
Qwen3-Coder-480B	34.5	46.6	40.5
Claude	46.1	45.7	45.9
SFT	46.1	56.1	51.1
COT Models			
SFT-CoT	**42.3** / 49.7	**37.6** / 44.7	**39.9** / 47.2
CoT-Edit-Merge.Code-RM	**38.4** / 29.8	**29.9** / 29.6	**34.1** / 28.2
CoT-Edit-Merge.Base	**55.9** / 55.3	**64.5** / 64.2	**60.2** / 59.8
CoT-Edit-Merge.Chat	**51.8** / 53.6	**58.8** / 56.7	**55.3** / 55.1

Table 1: Offline evaluation results of different models. All results are measured by EMR (Exact Match Ratio, %). For COT models, results are reported as: **COT inference /** Non-COT inference. Underlined values indicate the best performance.

Model	AR
Zeta	7.1%
SFT	16.95%
CoT-Edit-Merge.Base	19.36%

Table 2: Online A/B testing results. Zeta is the control group (strong baseline model). SFT and CoT-Edit-Merge.Base are our two treatment groups.

our data construction pipeline effectively captures real user behavior distribution.

RQ2: ***Does reasoning hurt or help with SFT?*** Comparing SFT and SFT-CoT in Table 1, we observe that SFT-CoT underperforms SFT under both inference modes: non-CoT inference (39.9% vs. 51.1%, -11.2pp) and CoT inference (47.2% vs. 51.1%, -3.9pp). This gap indicates that naively adding multi-step reasoning and training it with supervised fine-tuning alone does not automatically translate into better reasoning. Instead, it introduces additional optimization difficulty and error modes in the reasoning trajectory, which supervised fine-tuning alone cannot effectively correct. This observation motivates our adoption of reinforcement learning, which uses reward signals to correct reasoning errors.

RQ3: ***How does RL unlock CoT reasoning capabilities?*** In contrast to SFT-CoT's degradation, our RL-based approach successfully elicits reasoning capabilities through reward-driven optimization. As shown in Table 1, both RL-CoT variants substantially outperform SFT-CoT under CoT inference (CoT-Edit-Merge.Base: 60.2% vs. 47.2%, +13.0pp; CoT-Edit-Merge.Chat: 55.3% vs. 47.2%, +8.1pp), and both surpass the vanilla SFT baseline (51.1%). This improvement stems from RL's ability to assign credit to intermediate reasoning steps: RL algorithms leverage reward signals to explicitly reinforce correct reasoning paths and penalize error-prone trajectories. Notably, the Base variant achieves the strongest performance (60.2%), demonstrating that reward-driven optimization effectively guides the model toward more reliable multi-step reasoning patterns.

RQ4: ***Can LoRA-merge mechanism truly enhance performance?*** Comparing CoT-Edit-Merge.Base and CoT-Edit-Merge.Chat in Table 1, we observe a surprising phenomenon: the LoRA-merge mechanism substantially improves performance, with the Base variant achieving 60.2% accuracy compared to Chat's 55.3% (+4.9pp). This gain *suggests* that merging RL-trained LoRA adapters into code-base models *may* effectively combine the strengths of both model families.

RQ5: ***What is the cost of switching from CoT to non-CoT mode?*** As described in §3.2.1, our CoT models support seamless switching between CoT and non-CoT inference modes. Comparing the two modes in Table 1, we observe that CoT-Edit-Merge.Base achieves 60.2% under CoT inference and 59.8% under non-CoT inference, with only a modest degradation of 0.4pp. Similarly, CoT-Edit-Merge.Chat maintains strong performance (CoT: 55.3% vs. non-CoT: 55.1%, -0.2pp). This minimal performance drop demonstrates that our RL-trained models preserve their reasoning capabilities even when intermediate steps are suppressed. More importantly, switching to non-CoT mode dramatically improves efficiency: inference latency drops from 1.8s to 0.45s (4× speedup), making the model

practical for production deployment. This flexibility—balancing accuracy and efficiency through mode switching—validates the effectiveness of our RL-CoT approach in real-world scenarios.

RQ6: ***Are multi-step prompt and reward design essential for RL training?*** Figure 8 shows the model convergence across four configurations. The top-left panel shows successful convergence of our full approach (multi-step prompt + edit-aware reward), with the reward curve steadily increasing and stabilizing. In contrast, the top-right panel demonstrates that using non-CoT prompts leads to training failure, with oscillating curves indicating the model cannot learn effective editing strategies without structured reasoning. The bottom-left panel shows that replacing our edit-aware reward with overall code similarity rewards leads to deceptive convergence: the reward curve appears to increase rapidly (exceeding 0.5 within 100 steps), yet Table 1 reveals that CoT-Edit-Merge.Code-RM achieves only 34.1% EMR compared to CoT-Edit-Merge.Base's 60.2%. This stark performance gap validates the effectiveness of our edit-centric reward design. Finally, the bottom-right panel shows that LLM-as-a-Judge rewards fail to converge due to the judge model's limited reasoning capacity, which introduces noise into the RL reward signal and hinders effective learning. These ablations confirm that both the multi-step prompt (structuring the reasoning process) and the edit-aware reward design (providing precise modification feedback) are essential for effective RL training.

4.3 Case Study

As previously discussed, our core objectives are: **(1) handling edits that require deep semantic understanding beyond superficial patterns**, and **(2) improving interpretability in edit decisions**. Here we demonstrate the effectiveness of our proposed RL-CoT method through representative cases. We categorize edits requiring deep semantic understanding into two types: **(1) Insufficient Context**, **(2) Strong Semantic Dependency**. We select authentic user contexts from production data and compare predictions from our model CoT-Edit-Merge.Base against strong baseline Zeta. To validate interpretability, we evaluate CoT-Edit-Merge.Base in CoT-inference mode to expose its reasoning process.

Case 1 (see Figure 9 in Appendix G.1) demonstrates a scenario where the developer adds a `return` statement to handle a fallback case when `Globals.editor_context.const_list` is not available, but leaves the return value unspecified. Critically, the surrounding context provides no explicit type information for the expected return value. Zeta ignores this uncertainty and rashly generates the sophisticated completion `return [] if isinstance(value, (list, tuple)) else ""`, which is semantically questionable without concrete type evidence. In contrast, our model recognizes the insufficient context through its reasoning process and prudently chooses not to edit, demonstrating a safer approach that avoids introducing potentially incorrect assumptions when critical information is missing.

Case 2 (see Figure 10 in Appendix G.2) shows a scenario where the developer adds a condition `and :` to an existing `if` statement that controls `beidou_topic_app_id.append(game_id)`. The completion requires understanding the semantic link between the condition and the list operation. Zeta performs generic deduplication logic without understanding the actual intent. In contrast, our model correctly identifies that the condition guards the append operation by checking `game_id` validity, demonstrating deep semantic understanding.

4.4 Analysis of LoRA Merge Mechanism

As shown in §4.2.1 (RQ4), merging RL-trained LoRA adapters into the base model yields a +4.9pp improvement over merging into the chat model. We attribute this phenomenon to two factors:

(1) Weight Space Homogeneity: (see Figure 11 in Appendix H.1) Layer-wise analysis reveals ≥97% cosine similarity between chat and base model weights, ensuring that adapters trained on W_{chat} remain in a valid subspace when applied to W_{base}.

(2) Functional Orthogonality: (see Figure 12 in Appendix H.2) Cosine similarity analysis suggests that the RL-trained adapter and the corresponding base-model component lie in approximately orthogonal directions (Zhang and Zhou, 2025; Wang et al., 2023), enabling additive composition with minimal interference. Correspondingly, the adapter specializes in instruction-following and edit logic, whereas the base model retains strong code completion ability.

5 RELATED WORK

Transformer-based Code Edits. Several transformer-based approaches extend code generation to code editing (Chen et al., 2021; Gupta et al., 2023; Chakraborty et al., 2022; Liu et al., 2024a; Wang et al., 2021; Feng et al., 2020; Zhang et al., 2022). However, framing edit generation as translation limits their ability to improve user experience.
Instruction-based Models. These approaches introduce instructions for code edit (Li et al., 2024; Team and et al., 2025; Austin et al., 2021; Chen et al., 2021; Huang et al., 2025), but lack optimization for the stricter requirements of low latency and efficiency.
Pattern-based Models. Existing works (Zed, 2025; Liu et al., 2024b; Chen and et al., 2025; Wei et al., 2024; Wang et al., 2025) learn from historical edit behaviors to provide style-consistent, instruction-free suggestions that enable predictive collaboration. Building on this foundation, we advance pattern-based approaches by incorporating explicit reasoning, enabling models to demonstrate deep semantic understanding through interpretable steps before generating edits.

6 Conclusion

In this paper, we introduce a reinforcement learning paradigm that formulates code edit as a multi-step reasoning task. By integrating multi-step reasoning framework and proposing an edit-aware reward function, our method delivers grounded and inspectable reasoning, especially in complex and ambiguous code edit scenarios. Furthermore, we introduce a LoRA-merge mechanism that transfers reasoning capabilities from code-chat models to code-base models, enhancing generalization. Extensive offline evaluations and online A/B tests demonstrate that our approach achieves significant improvements across key metrics, providing a practical solution for industrial code edit suggestion.

7 Limitations

Language Coverage. Our evaluation focuses exclusively on Python and Java. While these are among the most widely used languages, the effectiveness of our approach on languages with substantially different syntax and semantics (e.g., Rust, Haskell) remains to be validated.
Benchmark Availability. Public benchmarks for industrial-scale code editing are currently lacking, limiting direct comparison with deployment environments.

References

Anthropic. The claude 3 model family: Opus, sonnet, haiku.

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

Stuart K Card. 2018. *The psychology of human-computer interaction*. Crc Press.

Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2022. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, 48(4):1385–1399.

Mark Chen and 1 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Xinfang Chen and et al. 2025. An efficient and adaptive next edit suggestion framework with zero human instructions in ides. *Preprint*, arXiv:2508.02473.

Copilot. 2023. Github copilot.

Cursor. 2024. Cursor.

DeepSeek-AI and et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. *Preprint*, arXiv:2002.08155.

Priyanshu Gupta and 1 others. 2023. Grace: Language models meet code edits. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1483–1495.

Siming Huang, Tianhao Cheng, J. K. Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J. Yang, Jiaheng Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Zhaoxiang Zhang, Jie Fu, Qian Liu, Ge Zhang, Zili Wang, Yuan Qi, Yinghui Xu, and Wei Chu. 2025. Opencoder: The open cookbook for top-tier code large language models. *Preprint*, arXiv:2411.04905.

Binyuan Hui and et al. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Kaixin Li, Qisheng Hu, Xu Zhao, Hui Chen, Yuxi Xie, Tiedong Liu, Qizhe Xie, and Junxian He. 2024. Instructcoder: Instruction tuning large language models for code editing. *Preprint*, arXiv:2310.20329.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. 2025. Limr: Less is more for rl scaling. *Preprint*, arXiv:2502.11886.

Changshu Liu, Pelin Cetin, Yogesh Patodia, Saikat Chakraborty, Yangruibo Ding, and Baishakhi Ray. 2024a. Automated code editing with search-generate-modify. *Preprint*, arXiv:2306.06490.

Chenyan Liu, Yufan Cai, Yun Lin, Yuhuan Huang, Yunrui Pei, Bo Jiang, Ping Yang, Jin Song Dong, and Hong Mei. 2024b. Coedpilot: Recommending code edits with learned prior edit relevance, project-wise awareness, and interactive nature. *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*.

Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hridesh Rajan. 2013. A study of repetitiveness of code changes in software evolution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 180–190.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *Preprint*, arXiv:2402.03300.

Kimi Team and et al. 2025. Kimi k2: Open agentic intelligence. *Preprint*, arXiv:2507.20534.

Peiding Wang, Li Zhang, Fang Liu, Yinghao Zhu, Wang Xu, Lin Shi, Xiaoli Lian, Minxiao Li, Bo Shen, and An Fu. 2025. Efficientedit: Accelerating code editing via edit-oriented speculative decoding. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 2619–2630.

Xiao Wang, Tianze Chen, Qiming Ge, Han Xia, Rong Bao, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. 2023. Orthogonal subspace learning for language model continual learning. *Preprint*, arXiv:2310.14152.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

Jiayi Wei, Greg Durrett, and Isil Dillig. 2024. Coeditor: Leveraging repo-level diffs for code auto-editing. In *The Twelfth International Conference on Learning Representations*.

An Yang and et al. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Zed. 2025. Zeta.

Haobo Zhang and Jiayu Zhou. 2025. Unraveling LoRA interference: Orthogonal subspaces for robust model merging. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 26459–26472, Vienna, Austria. Association for Computational Linguistics.

Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022. Coditt5: Pretraining for source code and natural language editing. *Preprint*, arXiv:2208.05446.

A Difference Modeling

Algorithm 1 Difference Modeling

Require: A stream of edit events E_{stream}, where each event $e = (C_{\text{pre}}, C_{\text{post}}) \in E_{\text{stream}}$ encompasses the pre- and post-modification content of code snippets

Ensure: Edit history $\mathcal{H}$

1: $\mathcal{H} \leftarrow \emptyset, \Delta \leftarrow \texttt{null}$
2: **for** $(C_{\text{pre}}, C_{\text{post}}) \in E_{\text{stream}}$ **do**
3:     $\Delta' \leftarrow \text{Diff}(C_{\text{pre}}, C_{\text{post}})$
4:     **if** $\Delta = \texttt{null}$ **or** $\neg\text{Overlap}(\Delta, \Delta')$ **then**
5:         **if** $\Delta \neq \texttt{null}$ **then**
6:             $\mathcal{H} \leftarrow \mathcal{H} \cup \{\Delta\}$
7:         **end if**
8:         $\Delta \leftarrow \Delta'$
9:     **else**
10:         $\Delta \leftarrow \text{Merge}(\Delta, \Delta')$
11:     **end if**
12: **end for**
13: **if** $\Delta \neq \texttt{null}$ **then**
14:     $\mathcal{H} \leftarrow \mathcal{H} \cup \{\Delta\}$
15: **end if**
16: **return** $\mathcal{H}$

B Implementation Details

Supervised Fine-Tuning. We fine-tune each model for one epoch using 4 Nvidia H200 GPUs with a global batch size of 64. Training is performed using the AdamW optimizer with learning rate of 2×10^{-4}. The maximum sequence length is set to 4096 tokens.

Reinforcement Learning. We adopt the ms-swift framework[14] and perform training on 8 Nvidia H200 GPUs. The training is conducted with LoRA fine-tuning (rank=64, alpha=64, targeting all linear layers). The model is trained with a batch size of 8 per device and 8 response generations per prompt (group size=8 for GRPO). Training runs for 4000 steps with a learning rate of 5×10^{-5}, warm-up ratio of 0.03, and KL coefficient $\beta =$

0.001. The maximum input and output lengths are 5000 and 1024 tokens, respectively. We enable model and optimizer offloading to handle large-scale training. All learning rates follow a cosine decay schedule. The hyperparameters α and β of the correctness reward are 0.5 and 0.5, respectively.

C Derivation-Based Filtering Prompt

Figure 4 presents the complete prompt template designed to guide multiple LLMs (e.g., Claude, DeepSeek-V3, Qwen-Coder) in performing collaborative relevance filtering.

D Multi-step Framework Design

Figure 5 presents a comparative view of prompt designs: the left side shows the standard non-CoT prompt commonly used in existing approaches, while the right side demonstrates our proposed multi-step prompt that guides the model through structured reasoning.

E Diff-Based vs. Full-Code Similarity

This section compares diff-based similarity with full-code similarity. As shown in Figure 6, the ground truth edit modifies `def __init__(self, operator, right):` to `def __init__(self, operator: Token, right: Expression):`, adding type annotations. In contrast, the model prediction changes `def __init__(self, operator, right):` to `def __init__(self, left, operator, right):`, adding a new parameter instead. The diff-based method assigns an 88.5% similarity score, correctly identifying the distinct nature of the two edit operations. However, the full-code similarity method compares the entire code content of the ground truth and prediction, as shown in Figure 7, yielding a 97.5% score that incorrectly suggests the edits are nearly identical.

F Model Convergence Under Different Configurations

Figure 8 illustrates the convergence behavior of different models during RL training.

G Case Study

G.1 Insufficient Context Case

Figure 9 illustrates a scenario where the model lacks sufficient context to generate accurate edits.

G.2 Strong Semantic Dependency

Figure 10 demonstrates a case where the edit requires understanding strong semantic dependencies across different code regions. This highlights the necessity of explicit reasoning to comprehend the logical connections between related code segments before proposing modifications.

H LoRA Merge Mechanism

H.1 Weight Space Homogeneity

Figure 11 demonstrates that the weights of Qwen2.5-Coder-7B-Base and Qwen2.5-Coder-7B-Instruct are highly similar.

H.2 Functional Orthogonality

Figure 12 demonstrates that the LoRA modules trained on Qwen2.5-Coder-7B-Instruct are highly orthogonal to the weights of Qwen2.5-Coder-7B-Base.

relevance-filter-Prompt

You are a code analysis expert. Your task is to determine if a code edit is a **logically derivable** continuation of the developer's work pattern and code context.

Prior Edits:
This section details the user's historical modifications, serving as the editing trajectory leading up to the current state:{}

Context:
This section provides a larger, detailed excerpt of the code surrounding the user's current cursor position, representing the current code state and broader situational context:{}

Next Edit:
This section presents the recommended subsequent code modification that is being evaluated for derivability:{}

Guidelines
- **Purpose**: predict the user's next small, useful edit based on their prior edits.
- Keep every modification minimal and evidence-driven; completeness or perfect syntax is not required.
- Prefer completing what the user just typed over suggesting to delete what they typed.
- Do **not** alter any numeric literals.
– Do **not** insert blank lines.
– Do **not** add placeholders such as `pass`, `TODO`, etc.
– **NEVER** roll back any prior edit; if a rollback seems necessary, leave the line unchanged.

Workflow
Step1. Analyze prior edits and the code context: identify exactly what the user deleted, added, or replaced, and infer their inner intent.
Step2. Analyze the recommended next edit.
Step3. Determine if the recommended edit aligns with the user's intent and complies with the guidelines.
- If it does, reply: "RELEVANT"
- If it does not, reply: "IRRELEVANT"

Figure 4: Prompt for derivation-based filtering.

Direct-Prompt

Instruction:
You are an intelligent code completion assistant. Your task is to predict the next edit a developer will make within a specific code region by analyzing their recent changes and current cursor position.

Context:
- **User Edits**:{}
- **User Excerpt**:{}
......
Example structure:
<answer>
<|editable_region_start|>
[your suggested code here]
<|editable_region_end|>
</answer>

let me solve this problem.\n<answer>

non-CoT mode

Multi-Step-Prompt

Instruction:
You are an intelligent code completion assistant. Your task is to predict the next edit a developer will make within a specific code region by analyzing their recent changes and current cursor position.

Context:
- **User Edits**:{}
- **User Excerpt**:{}
......
<think>
1. Edit pattern analysis: [Analyze the edit history to understand the developer's intent and coding pattern]
2. Developer's intent: [Identify what logical next step the developer is likely taking]
3. Suggested next step: [Predict and complete the next edit within the editable region only]
</think>
......

Let me solve this step by step.\n<think>

CoT mode

Figure 5: Comparison between the standard non-CoT prompt (left) and our multi-step reasoning prompt (right).

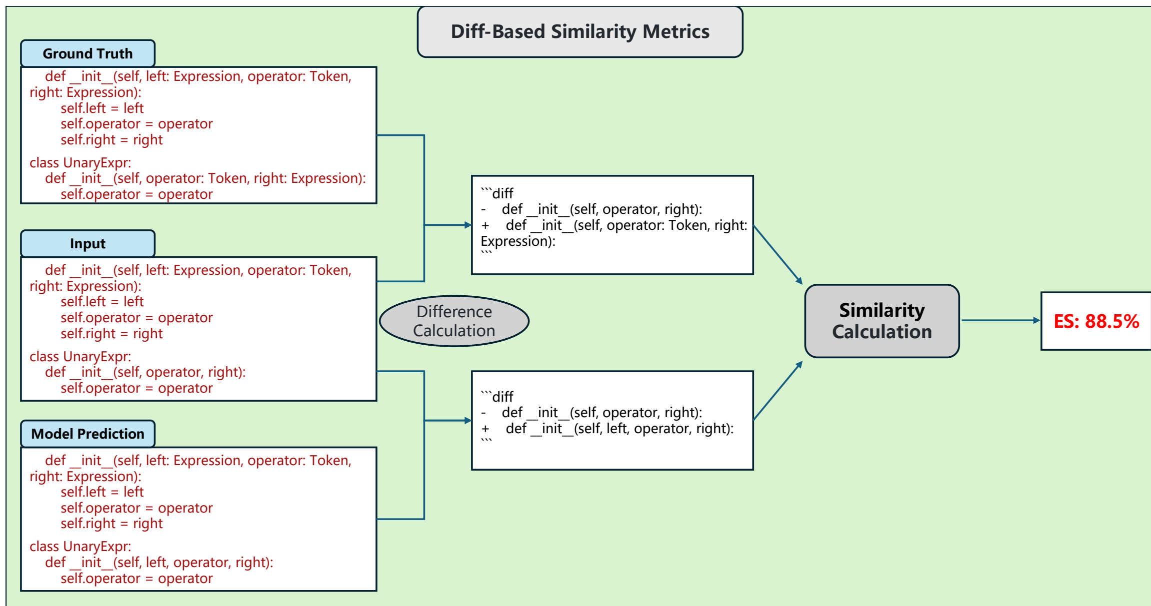

Figure 6: An example illustrating diff-based similarity.

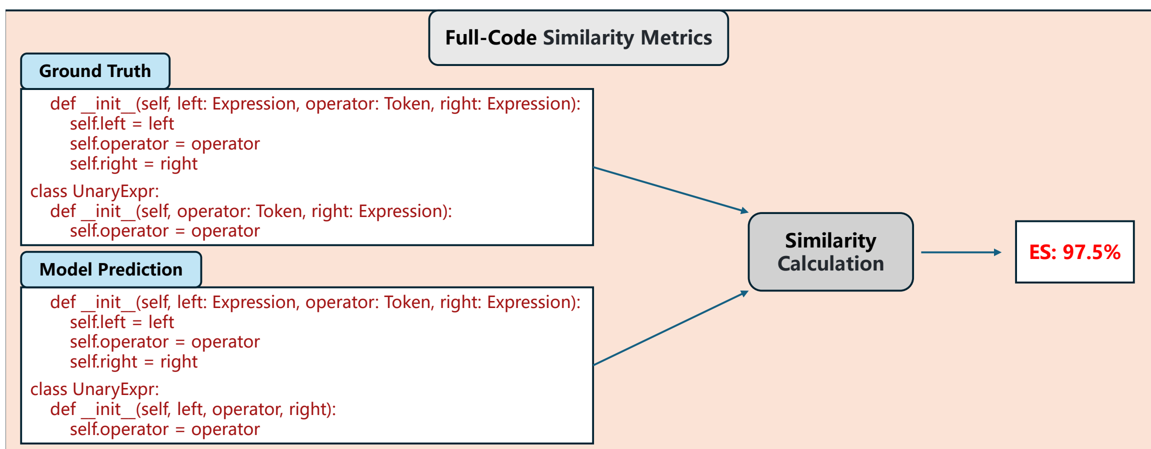

Figure 7: An example illustrating full code similarity.

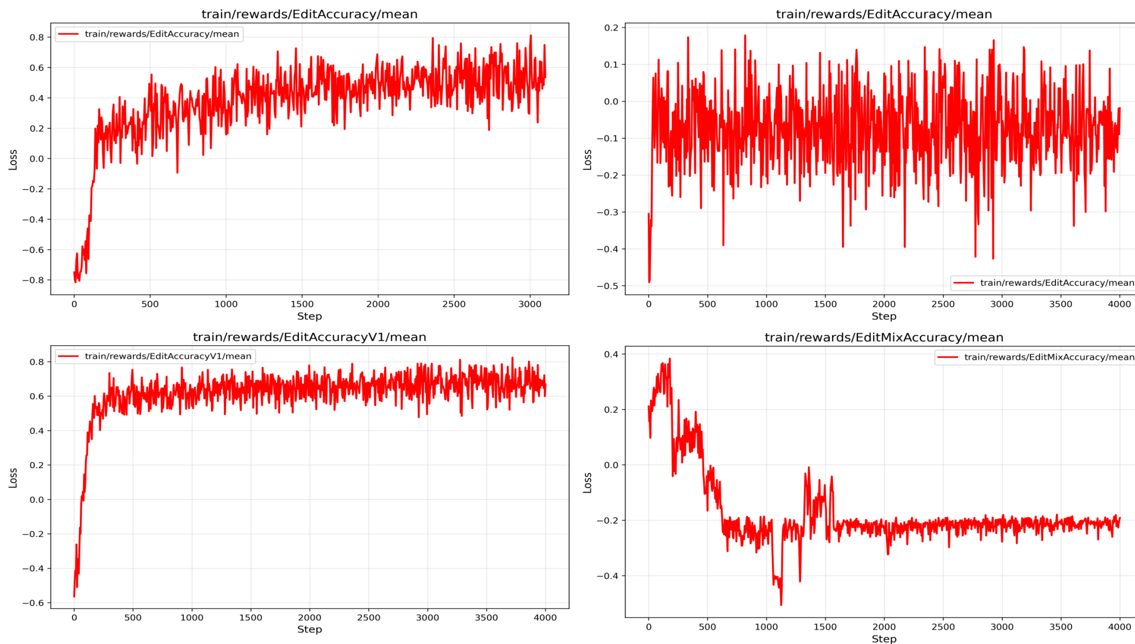

Figure 8: Model Convergence Under Different Configurations.

User-Edit

```
```diff
-
+                return
```
```

Context

```
def get_name_by_const_list(const_list_key, value):
    if is_run_on_pc_editor():
            const_list = Globals.const_list.get_const_list(const_list_key)
    else:
<|editable_region_start|>
        if not Globals.editor_context.const_list:
                return<|user_cursor_is_here|>
        const_list = Globals.editor_context.const_list.get_const_list(const_list_key)
    # 根据输入类型，在源列表为空时返回正确的空值
    if not const_list:
            return [] if isinstance(value, (list, tuple)) else ''
    # 优化：遍历一次嵌套列表，构建一个 key->name 的快速查找映射
    key_to_name_map = {}
<|editable_region_end|>
```

COT-MODEL OUTPUT

<think>

1. Edit pattern analysis: The developer is working on a function called `get_name_by_const_list`. They are handling different scenarios based on whether the code is running on a PC editor or not.
2. Developer's intent: It seems that the developer is trying to handle the case where `Globals.editor_context.const_list` is not available. They are likely preparing for a fallback scenario.
3. Suggested next step: The developer should handle the case where `const_list` is not available and return an appropriate value.

</think>

```
<|editable_region_start|>
        if not Globals.editor_context.const_list:
                return
        const_list = Globals.editor_context.const_list.get_const_list(const_list_key)
    # 根据输入类型，在源列表为空时返回正确的空值
    if not const_list:
            return [] if isinstance(value, (list, tuple)) else ''
    # 优化：遍历一次嵌套列表，构建一个 key->name 的快速查找映射
    key_to_name_map = {}
<|editable_region_end|>
```

Non-COT-MODEL OUTPUT

```
<|editable_region_start|>
        if not Globals.editor_context.const_list:
                return
                return [] if isinstance(value, (list, tuple)) else ''
        const_list = Globals.editor_context.const_list.get_const_list(const_list_key)
    # 根据输入类型，在源列表为空时返回正确的空值
    if not const_list:
            return [] if isinstance(value, (list, tuple)) else ''
    # 优化：遍历一次嵌套列表，构建一个 key->name 的快速查找映射
    key_to_name_map = {}
<|editable_region_end|>
```

Figure 9: Example case with insufficient context.

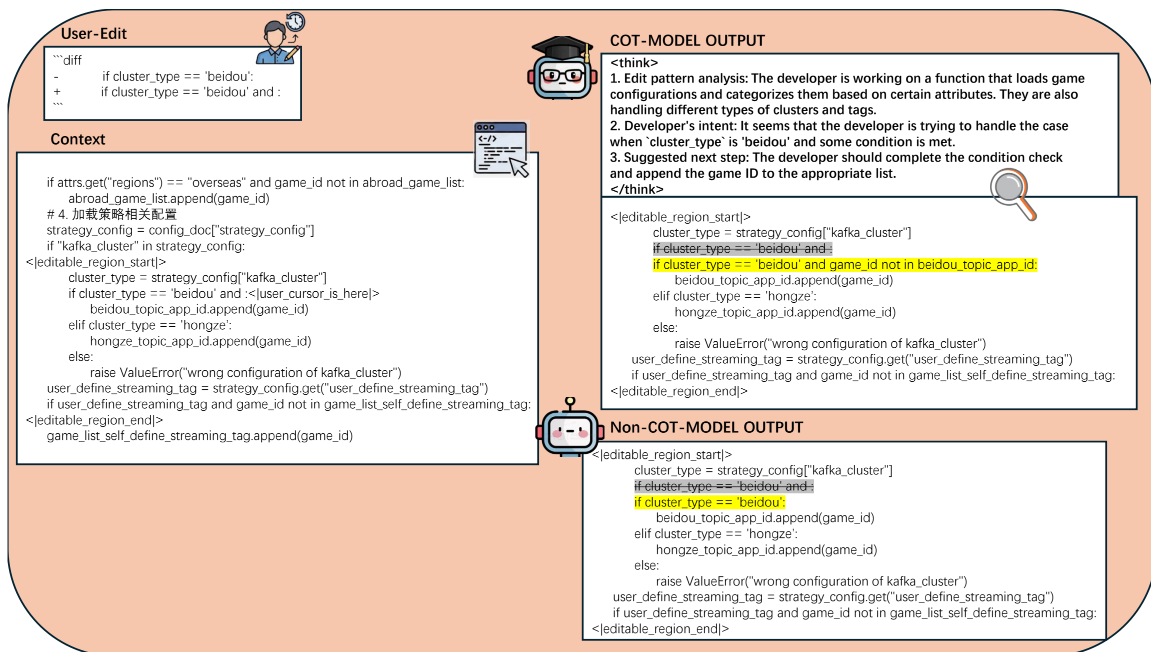

Figure 10: Example case with strong semantic dependency.

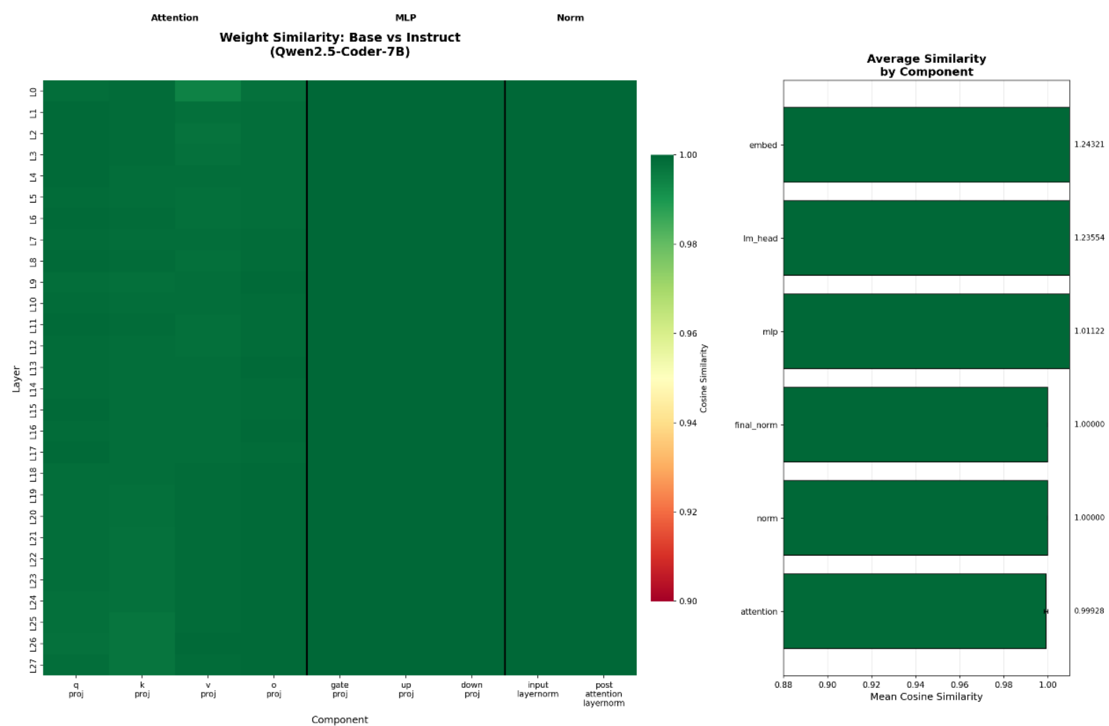

Figure 11: Cosine similarity analysis of weight differences between Qwen2.5-Coder-7B-Base and Qwen2.5-Coder-7B-Instruct.

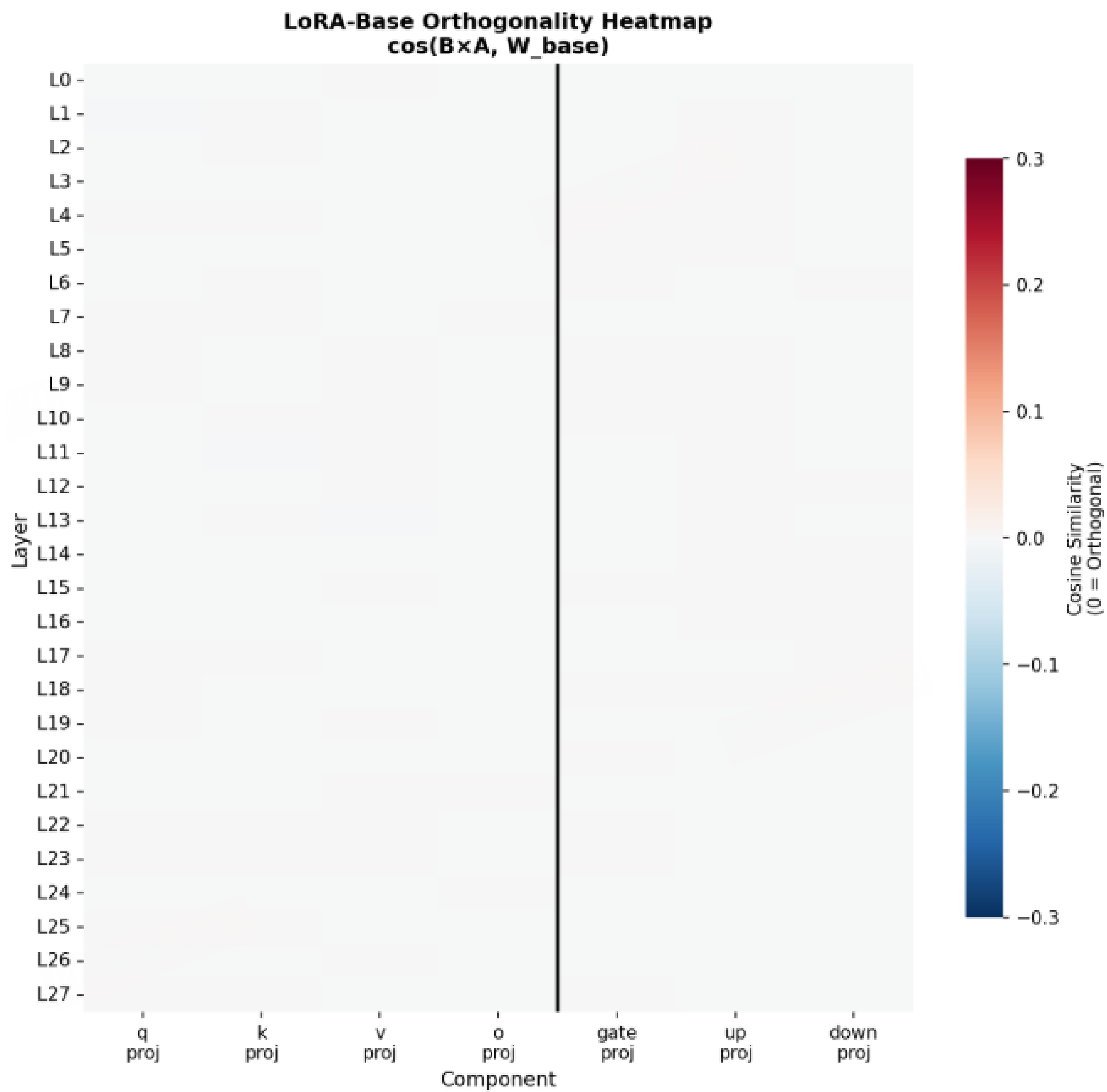

Figure 12: Cosine similarity analysis of weight differences between Qwen2.5-Coder-7B-Base and LoRA Modules.