To Diff or Not to Diff? Structure-Aware and Adaptive Output Formats for Efficient LLM-based Code Editing

Wei Cheng[1*], **Yongchang Cao**[2], **Chen Shen**[1], **Binhua Li**[2], **Jue Chen**[2], **Yongbin Li**[2†], **Wei Hu**[1†]
[1] State Key Laboratory for Novel Software Technology, Nanjing University, China
[2] Tongyi Lab, Alibaba Group, China
{wchengcs,cshen}.nju@gmail.com, whu@nju.edu.cn
{caoyongchang.cyc,binhua.lbh,chenjue.chen,shuide.lyb}@alibaba-inc.com

Abstract

Large Language Models (LLMs) are increasingly used for code editing, yet the prevalent full-code generation paradigm suffers from severe efficiency bottlenecks, posing challenges for interactive coding assistants that demand low latency and cost. Despite the predominant focus on scaling model capabilities, the edit format itself has been largely overlooked in model training. In this paper, we begin with a systematic study of conventional diff formats and reveal that fragile offsets and fragmented hunks make generation highly unnatural for LLMs. To address it, we introduce BLOCKDIFF and FUNCDIFF, two structure-aware diff formats that represent changes as block-level rewrites of syntactically coherent units such as control structures and functions. Furthermore, we propose ADAEDIT, a general adaptive edit strategy that trains LLMs to dynamically choose the most token-efficient format between a given diff format and full code. Extensive experiments demonstrate that ADAEDIT paired with structure-aware diff formats consistently matches the accuracy of full-code generation, while reducing both latency and cost by over 30% on long-code editing tasks.

1 Introduction

Large Language Models (LLMs) (Guo et al., 2024a; Hui et al., 2024) have become a cornerstone of modern software engineering (Dakhel et al., 2023; Kazemitabaar et al., 2023), where code editing that involves modifying source code according to an edit intent is one of fundamental tasks (Lientz et al., 1978; Fan et al., 2025). In Integrated Development Environments (IDEs), this task powers frequent, interactive features such as next edit suggestion (Cursor, 2023; Chen et al., 2025). While the research community has predominantly focused on scaling model capabilities to push the ceiling of edit accuracy, the strict efficiency constraints of real-world deployments are often sidelined. For real-time collaborative workflows, low latency and cost-saving are not merely optional enhancements, but mandatory prerequisites (Barke et al., 2023; Liang et al., 2024).

Consequently, the predominant *full-code generation* paradigm, where models always rewrite entire code for even minor edits, suffers from severe efficiency bottlenecks (Guo et al., 2024b; Singhal et al., 2024; Aggarwal et al., 2025). Even if the generation is accurate, this brute-force approach incurs high latency and prohibitive API costs by outputting vast amounts of redundant tokens, while simultaneously increasing the risk of unintended modifications (Zamfirescu-Pereira et al., 2025). Therefore, resolving this efficiency bottleneck without sacrificing edit accuracy represents a critical research challenge with immense industrial value.

Despite this pressing need, the *edit format* itself, i.e., the representation of code changes, has been largely overlooked as a pathway to efficiency. Current models either output full code or exhibit lazy coding by using informal placeholders, which lack the precision required for automated patching. Existing explorations (Gauthier, 2023a) of alternative edit formats, such as variants of the unified diff (Diffutils, 2002c) shown in Figure 1, are confined to prompting strategies for powerful instruction-following models. Consequently, the fundamental question of *how to design and adapt edit formats* in LLM-based code editing remains open.

In this paper, we first conduct a systematic study of conventional diff formats. Our findings reveal that number-indexed diff formats are highly fragile due to precise numerical offsets, while content-addressed diff formats mainly suffer from fragmented hunks that break the syntactic integrity of code. Generating these formats are fundamentally mismatched with the generative nature of LLMs, leading to substantially degraded edit accuracy.

[*] Work was partially done when interned at Tongyi Lab.
[†] Corresponding authors.

Findings of the Association for Computational Linguistics: ACL 2026, pages 29677–29694

```
@@ -5 +5 @@
-            process_user(user)
+            process_user(user, False)
```

(a) An example of MINUNIDIFF.

```
@@ ... @@
             print("Warning.")
-            process_user(user)
+            process_user(user, False)
             continue
```

(c) An example of MINCONTENTDIFF.

```
@@ -2,7 +2,7 @@
     for user in users:
         if not check_active(user):
             print("Warning.")
-            process_user(user)
+            process_user(user, False)
             continue

         if user.age >= 16:
```

(b) An example of UNIDIFF. The diff content of CONTENTDIFF is identical, but the line number header is replaced with `@@ ... @@`.

Figure 1: Examples of conventional diff formats. Refer to Figure 2 for the complete code editing task.

To respond to our findings, we introduce BLOCKDIFF and FUNCDIFF, two structure-aware diff formats that represent changes as block-level rewrites of syntactically coherent units, such as control structures and functions. By aligning textual diffs to code blocks extracted from the Abstract Syntax Tree (AST), we enable LLMs to edit in logical modules rather than arbitrary line fragments, which restores the naturalness of the generation process. However, any edit format is not universally optimal, where the overhead of diff formats can exceed that of full-code generation when changes are pervasive across the code. We therefore propose ADAEDIT, a general adaptive edit strategy that empowers LLMs to dynamically choose between a given diff format and full code. The model internalizes this switching logic during finetuning by trained on the most token-efficient format for each individual sample.

We evaluate different edit formats on diverse datasets (Zhang et al., 2025; Li et al., 2024; Muennighoff et al., 2024; Cassano et al., 2024; Gauthier, 2023b) and base models (Guo et al., 2024a; Hui et al., 2024). Our results demonstrate that ADAEDIT paired with structure-aware diff formats consistently matches full-code generation in edit accuracy, while reducing both latency and cost by over 30% on long-code editing tasks.

In summary, our main contributions are outlined as follows:

- We conduct a systematic study of conventional diff formats, revealing their fragile offsets and fragmented hunks for unnatural generation.
- We introduce two structure-aware diff formats that represent changes as block-level rewrites of syntactically coherent units.
- We propose ADAEDIT, a general adaptive edit strategy that trains LLMs to choose the most token-efficient format between a given diff format and full code.
- We conduct extensive experiments to demonstrate that ADAEDIT paired with structure-aware diff formats guarantees edit accuracy while significantly reducing latency and cost over full-code generation. Our source code is available at https://github.com/nju-websoft/AdaEdit.

2 Related Work

2.1 Code Editing with LLMs

Automated code editing initially focus on localized tasks like code completion and infilling (Izadi et al., 2022; Fried et al., 2023), which leverage Fill-in-the-Middle (FIM) techniques to predict missing regions (Bavarian et al., 2022). More recently, specialized models have emerged to perform non-trivial modifications without explicit location hints (Zhang et al., 2022; Li et al., 2023a). The advent of modern LLMs (Guo et al., 2024a; OpenAI, 2023; Hui et al., 2024; Team, 2025) has further unified these disparate tasks under a single generative framework, giving rise to what we term general-purpose code editing. It is defined as the task of modifying source code according to an edit intent, which may be expressed as explicit natural language instructions (Guo et al., 2024b; Singhal et al., 2024; Cassano et al., 2024), or implicit goals such as code repair and efficiency improvements (Zhang et al., 2023; Joshi et al., 2023; Wei et al., 2023; Chen et al., 2024b; Zhang et al., 2024; Olausson et al., 2024). Across this spectrum, a common paradigm of full-code generation has been adopted, leading most existing work to focus on enhancing the models' intrinsic capabilities to improve edit accuracy

Formats	EditEval	CanItEdit	HumanEvalFix	Aider-1	Aider-2	Average
Base model	55.39	42.98	65.12	35.56	44.44	48.70
FULLCODE	69.38	53.17	65.76	45.93	51.11	57.07
MINUNIDIFF	21.44	7.60	12.41	13.33	15.56	14.07
w/ numbers	41.49	23.93	42.80	20.74	26.67	31.13
UNIDIFF	48.07	17.33	39.60	28.89	31.85	33.15
w/ numbers	51.65	30.67	41.55	28.89	35.56	37.66
MINCONTENTDIFF	61.42	37.02	52.20	35.56	42.22	45.68
CONTENTDIFF	67.91	47.19	65.91	43.70	47.41	54.43

Table 1: Pass@1 comparison of conventional diff formats, trained on the Qwen2.5-Coder-7B base model.

(Muennighoff et al., 2024; Li et al., 2024; Aggarwal et al., 2025; Zhang et al., 2025). However, the fundamental role of the output format in determining edit efficiency has been largely overlooked.

2.2 Diff Formats and Their Applications

Text-based diff formats identify changes between plain text blocks and serve as the foundation of modern version control (Hunt and MacIlroy, 1976; Myers, 1986). However, the line-number indexing in these formats (Diffutils, 2002a,b,c) is inherently fragile for LLM generation since probabilistic models struggle with precise numerical offsets (CarperAI, 2023; Muennighoff et al., 2024). Alternatively, syntax-aware diff tools like GumTree represent modifications as symbolic operations on ASTs (Fluri et al., 2007; Falleri et al., 2014; Martinez et al., 2023; Falleri and Martinez, 2024). While effective for code analysis, these formats require specialized domain-specific languages that are fundamentally misaligned with the sequential text generation paradigm of LLMs. Furthermore, they typically ignore changes to non-semantic nodes, discarding modifications to spaces and even comments. This characteristic makes them entirely unsuitable for precise code generation and textual reconstruction. Recent studies (Gauthier, 2023a; Zamfirescu-Pereira et al., 2025) have explored content-addressed diff formats through sophisticated prompt engineering, relying on the powerful instruction-following capabilities of LLMs. In contrast, our work provides the first systematic training-based investigation into edit formats. Moreover, we operate under a completely different paradigm with existing diff tools, where we capture all textual modifications based on the standard unified diff and expand these exact diff hunks to encompass code structures. This ensures absolute textual fidelity while still leveraging structural context, culminating in the proposal of novel structure-aware formats and an adaptive edit strategy.

3 Challenges in Learning Diff Formats

3.1 Problem Formulation

To systematically investigate edit formats, we formalize LLM-based code editing as follows. Let $\mathcal{I}$ be the space of edit intents, $\mathcal{C}$ be the space of code snippets, and $\mathcal{E}$ be the space of edit representations.

Definition 1 (Edit Format)**.** An edit format is a pair of complementary functions, $\text{Diff} : \mathcal{C} \times \mathcal{C} \to \mathcal{E}$ and $\text{Patch} : \mathcal{C} \times \mathcal{E} \to \mathcal{C}$. For any source code $C \in \mathcal{C}$ and target code $C' \in \mathcal{C}$, these functions must satisfy the reconstruction identity:

$$\text{Patch}\left(C, \text{Diff}(C, C')\right) = C', \tag{1}$$

In this framework, full-code generation is a special case where $\text{Diff}(C, C') = C'$, while the Patch function serves as an identity projection.

Definition 2 (Edit Format Learning)**.** The goal of learning an edit format is to train a generative model $\mathcal{M}_\theta$ using a dataset $\mathcal{S} = \{(I_j, C_j, C'_j)\}_{j=1}^n$. Each triplet consists of an intent $I_j \in \mathcal{I}$, a source code $C_j \in \mathcal{C}$, and a target code $C'_j \in \mathcal{C}$. The learning process first transforms $\mathcal{S}$ into a training set $\mathcal{D} = \{(I_j, C_j, E_j)\}_{j=1}^n$ by computing $E_j = \text{Diff}(C_j, C'_j)$. Then, the model parameters are optimized by minimizing the objective:

$$\theta^* = \arg\min_\theta \sum_{j=1}^{n} \mathcal{L}\left(\mathcal{M}_\theta(I_j, C_j), E_j\right), \tag{2}$$

where $\mathcal{L}$ represents the token-level cross-entropy loss. During inference, $\mathcal{M}_{\theta^*}$ generates the specific edit representations for subsequent patching.

3.2 Number-Indexed Diff Formats

Following the experiment setup detailed in Section 5, we first investigate the widely-used unified diff format (Diffutils, 2002c) that relies on line

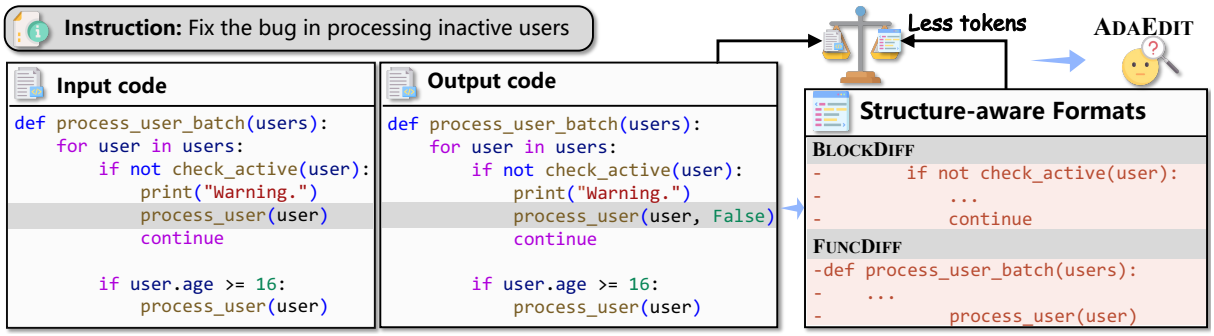

Figure 2: Overview of structure-aware diff formats and AdaEdit. Due to space constraints, lengthy code structures and added lines (+) are omitted in the diff examples.

numbers to locate changes. To improve robustness against malformed model outputs, we implement a custom patch tool that bypasses the standard consistency verification and operates solely on the specified line numbers. It includes two configurations:

- **MinUniDiff** is the minimal unified diff without context lines, as shown in Figure 1a.
- **UniDiff** is the standard unified diff with three context lines, as shown in Figure 1b.

For both formats, we also evaluate a "w/ numbers" variant, where the source code is augmented with line numbers to help LLM generation.

As depicted in Table 1, all number-indexed diff formats yield edit accuracies far below that of the full-code baselines (CarperAI, 2023; Muennighoff et al., 2024). This poor performance stems from the inherent fragility: LLMs struggle to generate precise line numbers and offsets, and this issue persists even when the input code is explicitly numbered.

3.3 Content-Addressed Diff Formats

To overcome the fragility of precise numerical offsets, content-addressed diff formats identify each edit region by a unique anchor content. They differ in the anchor content and follow a unified patching process detailed in Section 4.1. In this section, we include two variants:

- **MinContentDiff** is distinguished by its minimal anchor content. Each hunk's anchor is constructed by iteratively adding one surrounding context lines at a time, both above and below the edit position, until the resulting content is unique within the source code. An example is shown in Figure 1c.
- **ContentDiff** is based on MinContentDiff but requires as least three context lines, as shown in Figure 1b.

We also investigate different hunk styles in Appendix A, such as the unified diff-like and search/replace style (Gauthier, 2023a). Considering accuracy and robustness, we adopt the hunk rewrite style for all content-addressed diff formats.

The results in Table 1 reveal a crucial insight. While MinContentDiff outperforms all number-indexed diff formats, its accuracy remains substantially lags the full-code baselines. This indicates that eliminating line numbers is a necessary but insufficient step towards an effective edit format. While ContentDiff demands higher computational cost, this investment enhances the robustness of the anchor content, thereby significantly improving edit accuracy. Despite these gains, it still falls short of the finetuned full-code baseline.

We attribute the remaining performance gap to a form of fragmented hunks. In these conventional diff formats, the hunks consist of arbitrary line fragments. For an LLM extensively pre-trained on well-formed code, generating such disjointed snippets is fundamentally unnatural.

4 Methodology

The contrasting performance between full-code generation and conventional diff formats reveals a fundamental tradeoff between efficiency and naturalness. Full-code generation is inherently natural for LLMs since it aligns with their pre-training objectives, yet it suffers from extreme redundancy by re-generating unchanged code. In contrast, conventional diff formats aim to reduce token counts by targeting only modified segments, but their fragmented hunks are unnatural, which frequently compromises edit accuracy. Therefore, the central challenge lies in designing an edit format that achieves better efficiency than full-code generation while preserving sufficient naturalness for accuracy.

4.1 Structure-Aware Diff Formats

Unlike line-level changes in natural languages, code changes typically fall into syntactically cohesive units. Therefore, we introduce BLOCKDIFF and FUNCDIFF, two structure-aware diff formats that represent changes as block-level rewrites, as shown in Figure 2. By operating at different block levels, we hypothesize that they can restore sufficient naturalness for LLM generation.

Block Tree Construction We use `tree-sitter` (GitHub, 2018) to construct an AST from the source code. By identifying fine-grained AST nodes, we build a block tree including control structures (e.g., branches, loops, and contextual blocks) and functions. This hierarchy also incorporates coarse-grained nodes including classes and a `root` node representing the entire code. To maintain compatibility with unstructured segments and malformed code snippets, any contiguous lines outside fine-grained nodes, such as `import` statements and regions with syntax errors, are encapsulated into special fine-grained nodes. This strategy prevents local modifications from being improperly promoted to a coarse-grained parent. Based on this hierarchical structure, we define two distinct edit granularities. BLOCKDIFF permits edits at any fine-grained node level to maximize token efficiency, while FUNCDIFF ignores control structures to favor broader structural stability.

Diff Generation Given a source code and a target code, we first compute their MINUNIDIFF to cover all text diffs. Each diff hunk is then mapped to the block tree. A pure-insertion hunk is mapped to the smallest node that contains the insertion position. For a hunk that includes deletions, which may span multiple blocks, the target is the smallest set of contiguous block nodes that collectively contain all the deleted lines.

Although code blocks represent distinct units, their textual content is not always unique within the source code. To ensure unambiguous patching, we progressively expand the anchor content until it reaches contextual uniqueness. This process first incorporates immediately adjacent sibling nodes and then expands to the parent node if ambiguity persists. For each diff hunk, the expansion iterates until the selected anchor content is unique. In extreme cases where local uniqueness cannot be established, the expansion continues to the root node, thereby treating the entire code as the anchor.

Following the mapping and anchor expansion phases, individual hunks may contain overlapping lines. To eliminate redundant anchor content, we first merge any hunks that exhibit line-level overlaps. Furthermore, multiple distinct hunks may fall within the scope of a single fine-grained node, e.g., two branch changes in a loop. To enhance the naturalness of the generative process, we consolidate these hunks into their shared parent node. This transformation presents the modifications as a single logical edit instead of a series of fragmented changes. This structural merging proceeds iteratively in a bottom-up manner to ensure coherence across the entire block tree.

Patching The patching process is purely textual search and replacement operations without AST parsing. Given a source code and a content-addressed diff, each hunk is interpreted as locating the code region by its anchor content, then performing the specified deletion and insertion operations to transform the code. A patch failure occurs if the anchor cannot be uniquely located, resulting from either zero matches or multiple ambiguous matches. To mitigate occasional non-determinism in LLM outputs, the process incorporates a tolerance mechanism. It attempts to find a plausible match by progressively relaxing constraints on whitespace and blank lines when an exact textual match is unavailable (Gauthier, 2023a).

4.2 ADAEDIT: An Adaptive Strategy

The efficiency of diff formats is intrinsically tied to the magnitude of changes within the source code. Since these representations require both anchor and modified content, the token count can escalate when edits are extensive or dispersed across multiple regions. Under such conditions, the cumulative overhead of diff hunks may eventually exceed the cost of full-code regeneration. This creates a critical threshold where a diff format is no longer the most efficient choice for code editing, effectively negating the efficiency benefits typically associated with localized editing.

To ensure optimal efficiency across all scenarios, we propose ADAEDIT, a general adaptive edit strategy that trains LLMs to dynamically choose the most token-efficient format between a given diff format and full code. This switching logic is internalized during training through a data-driven proxy. For each pair of source code C_j and target code C'_j in the training set, we compare the lengths

Base models	Formats	EditEval	CanItEdit	HumanEvalFix	Aider-1	Aider-2	Average
DeepSeek-Coder-6.7B	Base model	40.03	29.62	42.65	27.41	37.04	35.35
	FULLCODE	64.38	**44.88**	56.95	45.19	49.63	52.21
	CONTENTDIFF	60.95	36.74	55.79	42.22	48.89	48.92
	w/ ADAEDIT	63.84	36.19	59.39	42.96	49.63	50.40
	BLOCKDIFF	**64.87**	34.74	58.02	44.44	51.11	50.64
	w/ ADAEDIT	64.69	38.98	**60.09**	45.19	51.85	52.16
	FUNCDIFF	64.25	35.86	59.79	44.44	49.63	50.79
	w/ ADAEDIT	64.79	38.95	59.76	**46.67**	**52.59**	**52.55**
Qwen2.5-Coder-7B	Base model	55.39	42.98	65.12	35.56	44.44	48.70
	FULLCODE	69.38	**53.17**	65.76	45.93	51.11	57.07
	CONTENTDIFF	67.91	47.19	65.91	43.70	47.41	54.43
	w/ ADAEDIT	68.02	48.60	65.88	44.44	49.63	55.31
	BLOCKDIFF	69.95	48.07	64.85	45.19	51.85	55.98
	w/ ADAEDIT	69.30	51.36	67.38	**47.41**	**52.59**	57.61
	FUNCDIFF	**70.88**	50.31	67.62	45.93	51.85	57.32
	w/ ADAEDIT	69.23	52.67	**67.84**	**47.41**	**52.59**	**57.95**
Qwen2.5-Coder-14B	Base model	58.25	46.67	71.52	41.48	51.11	53.81
	FULLCODE	69.59	**60.95**	70.40	54.07	**64.44**	63.89
	CONTENTDIFF	72.40	57.69	66.65	53.33	60.74	62.16
	w/ ADAEDIT	70.21	55.07	69.18	54.07	62.22	62.15
	BLOCKDIFF	72.40	60.40	69.97	54.07	63.70	64.11
	w/ ADAEDIT	71.47	60.48	69.88	54.07	63.70	63.92
	FUNCDIFF	**73.48**	58.64	**73.05**	**54.81**	**64.44**	**64.89**
	w/ ADAEDIT	71.52	60.21	72.41	**54.81**	**64.44**	64.68

Table 2: Main comparison of pass@1. The best results are marked in **bold**, and the second best are underlined.

of the diff representation and the full target code. We then re-define the ground truth E_j as the shorter format for that instance:

$$E_j = \underset{S \in \{C'_j, \text{Diff}(C_j, C'_j)\}}{\arg\min} |S|, \tag{3}$$

where $|\cdot|$ denotes the token count operator defined by the specific tokenizer of LLMs. By finetuning on this optimized dataset, the model implicitly learns to predict the more efficient format at inference time based on the provided editing task.

5 Experiments and Results

5.1 Datasets

Given the availability of high-quality code editing resources, we primarily focus on Python. Our main training dataset is **OCEData** (Zhang et al., 2025). We evaluate on four diverse Python benchmarks. These include two instruction-guided benchmarks that aligns with the training task: **EditEval** (Li et al., 2024) and **CanItEdit** (Cassano et al., 2024). We also incorporate two bug-fixing benchmarks that involve implicit edit intents: **HumanEvalFix** (Muennighoff et al., 2024), a test-driven repair task, and **Aider** (Gauthier, 2023b), which simulates realistic two-stage coding exercises involving initial generation and subsequent test-driven repair.

To demonstrate the generalizability, we also evaluate on another Python training dataset **InstructCoder** (Li et al., 2024). For other programming languages, we train on the JavaScript subset of **CommitPackFT** (Muennighoff et al., 2024) and evaluate on HumanEvalFix-JavaScript. More details are presented in Appendix B.1 and Table 6.

5.2 Implementation Details

We compare diff formats against two full-code baselines: the original non-finetuned **Base** model, and **FULLCODE** that serves as the upper bound for naturalness and accuracy. The representative conventional format CONTENTDIFF is also included for comparison with our structure-aware formats.

Our experiments primarily utilize the Qwen2.5-Coder series (Hui et al., 2024), specifically 7B and 14B, which represent the state-of-the-art in open-source code LLMs. We also evaluate on DeepSeek-Coder-6.7B (Guo et al., 2024a), an earlier model with different architectural variants. All models undergo full-parameter Supervised Fine-Tuning (SFT) from their base versions to ensure that performance variations stem solely from the choice of edit format. We apply a unified prompt template with format-specific prefixes across all evaluations. More details are shown in Appendix B.2.

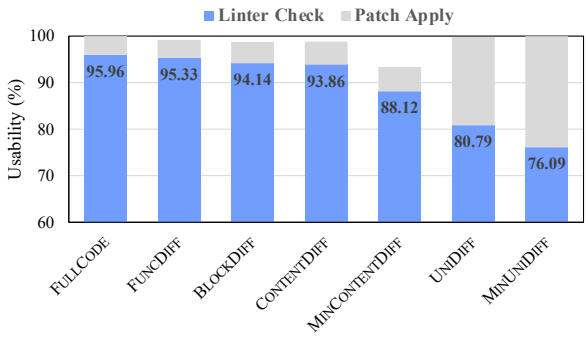

Figure 3: Edit usability comparison, trained on Qwen2.5-Coder-7B. The source code for MINUNIDIFF and UNIDIFF is augmented with line numbers.

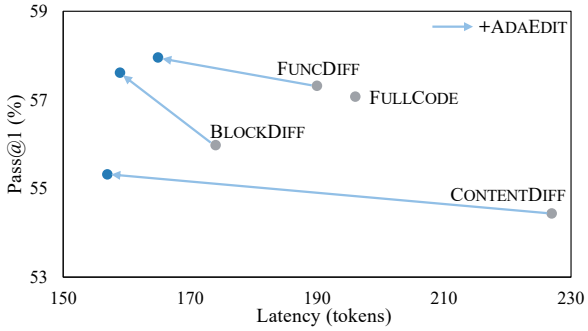

Figure 4: Latency-accuracy landscape of different edit formats and ADAEDIT, trained on Qwen2.5-Coder-7B.

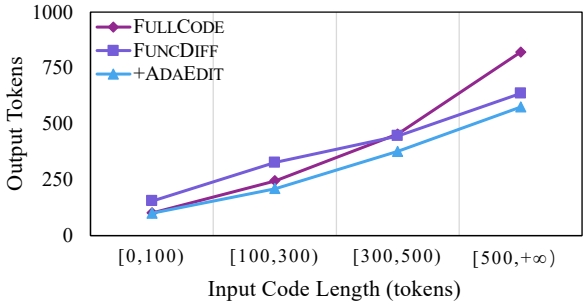

Figure 5: Edit cost comparison across code scales, trained on Qwen2.5-Coder-7B.

5.3 Evaluation Metrics

We assess each edit format from two primary perspectives and report the average of each metric. See Appendix B.3 for more details:

- **Effectiveness.** We evaluate edit accuracy by pass@1, and usability by exact patch-apply success rate and stricter linter check.
- **Efficiency.** We evaluate edit latency by tokens of the first renderable generation, and cost by tokens of the complete generation.

For ADAEDIT, we further report the percentage of samples where the model correctly chooses the most token-efficient edit format.

5.4 Effectiveness: Edit Accuracy & Usability

The main results in Table 2 demonstrate that BLOCKDIFF and FUNCDIFF consistently outperform the line-level CONTENTDIFF across all base models. By preserving syntactic coherence of diff hunks, structure-aware formats significantly restore the naturalness for LLM generation. We observe that FUNCDIFF consistently achieves superior accuracy among diff formats. With the more capable Qwen2.5-Coder models, the average accuracy of FUNCDIFF even exceeds the FULLCODE baseline. A similar phenomenon occurs with BLOCKDIFF when trained on Qwen2.5-Coder-14B, which suggests that the advantages of structural representations require more powerful models to fully realize.

The integration of ADAEDIT further enhances accuracy by enabling models to dynamically choose between a specific diff format and full-code generation. Although ADAEDIT is specifically optimized for token efficiency, the reduction in test-time inference costs does not generally compromise edit accuracy. This adaptivity is especially effective for complex tasks such as CanItEdit and Aider, demonstrating that a purely diff-based approach is not always the optimal solution. For Qwen2.5-Coder-14B, its superior capacity allows for generating effective diff representations even in complex tasks while fully leveraging test-time scaling on function-level benchmarks like EditEval (Hoffmann et al., 2022). In such instances, ADAEDIT maintains accuracy while significantly reducing inference costs, as evidenced in Section 5.5. Furthermore, structure-aware diff formats still consistently outperform CONTENTDIFF when integrated with ADAEDIT.

Except for functional correctness, we assess the basic usability of generated edits, as illustrated in Figure 3. Only CONTENTDIFF frequently encounters patch application failures, since the model is more likely to generate minimal anchor content that is not unique within the source code. In contrast, while number-indexed formats maintain nearly perfect patch success rates due to the absence of strict consistency verification, subsequent linter checks reveal significant damage to the resulting code. When evaluated under stricter linter checks, the observed usability trends closely mirror edit accuracy, where structure-aware formats consistently achieve superior usability among diff formats.

Formats	Pass@1 (%)	Cost (tokens)
FULLCODE	39.75	648.30
CONTENTDIFF	33.75	612.85
w/ ADAEDIT	33.00	**432.73**
BLOCKDIFF	38.69	570.26
w/ ADAEDIT	37.94	466.04
FUNCDIFF	**40.75**	546.77
w/ ADAEDIT	40.69	481.63

Table 3: Performance comparison on the CanItEdit subset containing 80 samples whose input code length is over 300 tokens, trained on Qwen2.5-Coder-7B.

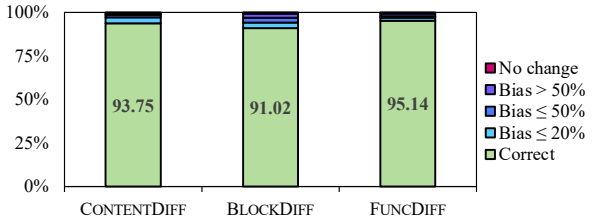

Figure 6: The accuracy of the format selection mechanism within ADAEDIT, trained on Qwen2.5-Coder-7B.

5.5 Efficiency: Edit Latency & Cost

For interactive coding assistants, both latency and accuracy are critical user-centric metrics. As illustrated in Figure 4, we observe that purely diff-based formats do not provide a substantial latency advantage, especially CONTENTDIFF whose average latency even exceeds that of FULLCODE. This counter-intuitive result is largely attributed to the characteristics of current benchmarks, which primarily consist of short, function-level samples. In such contexts, the anchor content required by diff formats constitutes a high fraction of the total token count. In contrast, ADAEDIT consistently shifts toward the top-left quadrant of the accuracy-latency spectrum, which not only reduces overall edit latency but also improves accuracy.

To dissect efficiency scaling properties, we analyze inference cost measured in the total output tokens. As depicted in Figure 5, the cost analysis reveals three key insights: (i) The output tokens of FULLCODE are directly proportional to the code length, making its cost prohibitive for long code. (ii) The overhead from function-level anchors makes FUNCDIFF inefficient on short code, but its efficiency gradually surpasses FULLCODE for longer code. (iii) ADAEDIT intelligently switches between both formats, which achieves optimal efficiency across all code scales and reduces cost by over 30% on long-code editing tasks.

As efficiency gains are more significant for input

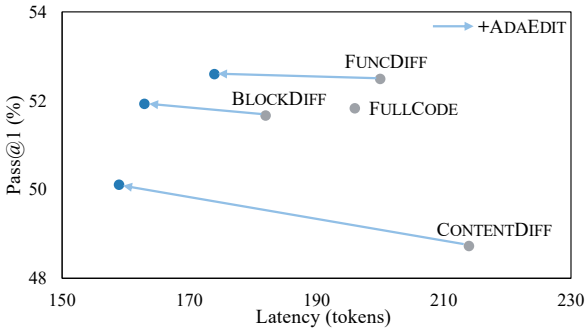

Figure 7: Latency-accuracy landscape, trained on InstructCoder with Qwen2.5-Coder-7B.

code exceeding 300 tokens, we analyze accuracy and cost specifically for this subset of CanItEdit. The results in Table 3 indicate that despite a marked reduction in computational cost, edit accuracy does not suffer a significant decline for structure-aware formats. Notably, FUNCDIFF and its combination with ADAEDIT even exhibit marginal accuracy improvements, which suggests a promising solution to the tension between accuracy and efficiency in long-code editing scenarios.

5.6 Analysis of the Adaptive Strategy

While ADAEDIT demonstrates strong performance in both effectiveness and efficiency, we further isolate and evaluate the reliability of its format selection mechanism using comprehensive categories. The "No change" category denotes instances where the model produces no valid edits. This typically occurs when the model directly echoes the input code without modifications, or generates invalid diffs. Furthermore, the "Bias" categories quantify the degree of token count deviation when the model makes an incorrect format selection.

Figure 6 shows that the accuracy of selecting the most token-efficient format exceeds 90% across all diff formats. When allowing for a slight deviation of 20% in token counts, the average accuracy surpasses 95%. Furthermore, we observe that selection correctness is closely related to the complexity of the diff representation. Simpler formats typically yield higher selection accuracy, whereas the decision-making burden is heavier for BLOCKDIFF due to its inclusion of diverse control structures.

To further validate the necessity of our adaptive strategy, we evaluate stronger LLMs (DeepSeek-AI, 2025; OpenAI, 2026) under a few-shot, inference-only setting. As detailed in Appendix E, these advanced models exhibit poor format selection accuracy and severe bias. These contrasting

Formats	HumanEvalFix-JavaScript
Base model	63.48
FULLCODE	66.13
CONTENTDIFF	56.55
w/ ADAEDIT	64.97
BLOCKDIFF	62.44
w/ ADAEDIT	65.70
FUNCDIFF	63.84
w/ ADAEDIT	**67.74**

Table 4: Pass@1 comparison, trained on the JavaScript subset of CommitPackFT with Qwen2.5-Coder-7B.

results confirm that LLMs do not inherently possess this cost-benefit logic, and that ADAEDIT is essential for effectively internalizing it.

5.7 Other Training Corpus

We conduct additional experiments using another Python training set, as shown in Figure 7 and detailed in Appendix F. Due to inherent differences in data quality, the absolute accuracy scores are lower than those observed with OCEData. However, the comparative relationships regarding accuracy and efficiency remain highly consistent across all edit formats. This stability demonstrates that the advantages of our designs can generalize effectively across different training data.

Our structure-aware formats can be readily extended to other programming languages such as JavaScript by simply adjusting the AST node configurations. The results in Table 4 demonstrate that BLOCKDIFF and FUNCDIFF exhibit an even more pronounced advantage over the line-level CONTENTDIFF in JavaScript. When integrated with ADAEDIT, the edit accuracy matches and even surpasses that of full-code generation. This highlights the cross-language robustness of our designs and the potential for multilingual code editing.

6 Conclusion

In this paper, we address the critical efficiency bottleneck in LLM-based code editing caused by the prevalent full-code generation paradigm. We begin with a systematic study that reveals the inherent unnaturalness of conventional diff formats for LLM generation. To resolve this, we introduce BLOCKDIFF and FUNCDIFF, two structure-aware diff formats that represent changes as block-level rewrites. Recognizing that any single format is not universally optimal, we then propose ADAEDIT, a general adaptive edit strategy that trains LLMs to dynamically choose the most token-efficient format for each editing task. Extensive experiments show that ADAEDIT paired with structure-aware diff formats reduces both latency and cost by over 30% on long-code editing tasks, while not compromising accuracy. Our work highlights the focus on the edit format itself and paves the way for more cost-effective coding assistants.

For future work, we identify two promising directions. First, while ADAEDIT establishes a robust cold-start foundation through supervised fine-tuning on the token-efficient pattern, incorporating reinforcement learning from verifiable rewards (Li et al., 2022) could allow models to autonomously explore and optimize intrinsic editing formats by balancing functional correctness and token efficiency in the reward function. Second, we plan to explore fluid granularity that transcends fixed AST boundaries. Although implementing semantically aggregated changes requires significantly more complex diff and patch architectures, it represents a highly valuable step toward more flexible and intelligent code editing.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62272219) and the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118).

Ethical Considerations

The datasets, benchmarks, and LLMs used in this work are public with permissive licenses.

Limitations

Our work has several limitations that also frame promising directions for future research.

The effectiveness of our proposed formats is closely tied to the capabilities of the base model. Unlike full-code generation, interpreting edit intention to produce a structure-aware diff or adaptively choosing the optimal format is more sophisticated. Our results confirm this dependency, as the performance benefits of our formats consistently scale with model size and capability.

Furthermore, our study is constrained by the general scarcity of high-quality, large-scale training data for complex code editing. Developing datasets for long-code and even repository-level

editing (Bairi et al., 2024; Jimenez et al., 2024) to scale edit formats remains a crucial undertaking.

As with all diff-based representations, including our structure-aware formats, there exist inherent risks of patch application failures or unintended code corruption. These potential issues necessitate careful validation when deploying such systems in real-world production environments. We mitigate these concerns in Section 5.4 by verifying this basic edit usability for structure-aware formats.

References

Tushar Aggarwal, Swayam Singh, Abhijeet Awasthi, Aditya Kanade, and Nagarajan Natarajan. 2025. NextCoder: Robust adaptation of code LMs to diverse code edits. In *ICML*, pages 1–23, Vancouver, Canada. PMLR.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, 2108.07732:1–34.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram K. Rajamani, Balasubramanyan Ashok, and Shashank Shet. 2024. CodePlan: Repository-level coding using LLMs and planning. *Proc. ACM Softw. Eng.*, 1(FSE):675–698.

Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How programmers interact with code-generating models. *Proc. ACM Program. Lang.*, 7(OOPSLA1):85–111.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *CoRR*, 2207.14255:1–30.

CarperAI. 2023. Diff model for code generation. https://huggingface.co/CarperAI/diff-codegen-6b-v2.

Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Anton Lozhkov, Carolyn Jane Anderson, and Arjun Guha. 2024. Can it edit? evaluating the ability of large language models to follow code editing instructions. In *COLM*, pages 1–46, Philadelphia, PA, USA.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *CoRR*, 2107.03374:1–35.

Songlin Chen, Weicheng Wang, Xiaoliang Chen, Peng Lu, Zaiyan Yang, and Yajun Du. 2024a. LLaMA-LoRA neural prompt engineering: A deep tuning framework for automatically generating chinese text logical reasoning thinking chains. *Data Intell.*, 6(2):375–408.

Xinfang Chen, Siyang Xiao, Xianying Zhu, Junhong Xie, Ming Liang, Dajun Chen, Wei Jiang, Yong Li, and Peng Di. 2025. An efficient and adaptive next edit suggestion framework with zero human instructions in ides. *CoRR*, 2508.02473:1–13.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024b. Teaching large language models to self-debug. In *ICLR*, pages 1–78, Vienna, Austria. OpenReview.net.

Cursor. 2023. Cursor. https://cursor.com.

Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. 2023. GitHub Copilot AI pair programmer: Asset or liability? *J. Syst. Softw.*, 203:111734.

DeepSeek-AI. 2025. DeepSeek-V3.2: Pushing the frontier of open large language models. *CoRR*, 2512.02556:1–23.

GNU Diffutils. 2002a. Detailed description of context format. https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Context.html.

GNU Diffutils. 2002b. Detailed description of normal format. https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Normal.html.

GNU Diffutils. 2002c. Detailed description of unified format. https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html.

Jean-Rémy Falleri and Matias Martinez. 2024. Fine-grained, accurate and scalable source differencing. In *ICSE*, pages 231:1–231:12, Lisbon, Portugal. ACM.

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ASE*, pages 313–324, Vasteras, Sweden. ACM.

Lishui Fan, Jiakun Liu, Zhongxin Liu, David Lo, Xin Xia, and Shanping Li. 2025. Exploring the capabilities of LLMs for code-change-related tasks. *ACM Trans. Softw. Eng. Methodol.*, 34(6):159:1–159:36.

Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. 2007. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743.

Python Software Foundation. 2018. Black: The uncompromising code formatter. https://github.com/psf/black.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A generative model for code infilling and synthesis. In *ICLR*, pages 1–26, Kigali, Rwanda. OpenReview.net.

Paul Gauthier. 2023a. Edit formats. https://aider.chat/docs/more/edit-formats.html.

Paul Gauthier. 2023b. GPT code editing benchmarks. https://aider.chat/docs/benchmarks.html#the-benchmark.

GitHub. 2018. Introduction to Tree-sitter. https://tree-sitter.github.io/tree-sitter.

Google. 2017. First Meaningful Paint. https://developers.google.com/web/tools/lighthouse/audits/first-meaningful-paint.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024a. DeepSeek-Coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, 2401.14196:1–23.

Jiawei Guo, Ziming Li, Xueling Liu, Kaijing Ma, Tianyu Zheng, Zhouliang Yu, Ding Pan, Yizhi Li, Ruibo Liu, Yue Wang, Shuyue Guo, Xingwei Qu, Xiang Yue, Ge Zhang, Wenhu Chen, and Jie Fu. 2024b. CodeEditorBench: Evaluating code editing capability of large language models. *CoRR*, 2404.03543:1–29.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, and 3 others. 2022. Training compute-optimal large language models. *CoRR*, 2203.15556:1–36.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *ICLR*, pages 1–16, Addis Ababa, Ethiopia. OpenReview.net.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder technical report. *CoRR*, 2409.12186:1–32.

James Wayne Hunt and M Douglas MacIlroy. 1976. *An algorithm for differential file comparison*. Bell Laboratories Murray Hill.

Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. CodeFill: Multi-token code completion by jointly learning from structure and naming sequences. In *ICSE*, pages 401–412, Pittsburgh, PA, USA. ACM.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can language models resolve real-world GitHub issues? In *ICLR*, pages 1–52, Vienna, Austria. OpenReview.net.

Harshit Joshi, José Pablo Cambronero Sánchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radicek. 2023. Repair is nearly generation: Multilingual program repair with LLMs. In *AAAI*, pages 5131–5140, Washington, DC, USA. AAAI Press.

Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI code generators on supporting novice learners in introductory programming. In *CHI*, pages 455:1–455:23, Hamburg, Germany. ACM.

Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, and Florian Bruhin. 2004. pytest 9.0.1. https://github.com/pytest-dev/pytest.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with PagedAttention. In *SOSP*, pages 611–626, Koblenz, Germany. ACM.

Jia Li, Ge Li, Zhuo Li, Zhi Jin, Xing Hu, Kechi Zhang, and Zhiyi Fu. 2023a. CodeEditor: Learning to edit source code with pre-trained models. *ACM Trans. Softw. Eng. Methodol.*, 32(6):143:1–143:22.

Kaixin Li, Qisheng Hu, James Xu Zhao, Hui Chen, Yuxi Xie, Tiedong Liu, Michael Shieh, and Junxian He. 2024. InstructCoder: Instruction tuning large language models for code editing. In *ACL-SRW*, pages 50–70, Bangkok, Thailand. Association for Computational Linguistics.

Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. 2023b. Sequence parallelism: Long sequence training from system perspective. In *ACL*, pages 2391–2404, Toronto, Canada. Association for Computational Linguistics.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2024. A large-scale survey on the usability of AI programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 52:1–52:13, Lisbon, Portugal. ACM.

Bennet P. Lientz, E. Burton Swanson, and G. E. Tompkins. 1978. Characteristics of applications software maintenance. *Commun. ACM*, 21(6):466–471.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *ICLR*, pages 1–19, New Orleans, LA, USA. OpenReview.net.

Yitian Luo, Yu Liu, Lu Zhang, Feng Gao, and Jinguang Gu. 2025. A survey on quality evaluation of instruction fine-tuning datasets for large language models. *Data Intell.*, 7(3):527–566.

Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. 2025. Lingma SWE-GPT: an open development-process-centric language model for automated software improvement. In *ISSTA*, pages 1–21, Trondheim, Norway. ACM.

Matias Martinez, Jean-Rémy Falleri, and Martin Monperrus. 2023. Hyperparameter optimization for AST differencing. *IEEE Trans. Software Eng.*, 49(10):4814–4828.

Michael McCloskey and Neal J Cohen. 1989. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. OctoPack: Instruction tuning code large language models. In *ICLR*, pages 1–59, Vienna, Austria. OpenReview.net.

Eugene W. Myers. 1986. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is self-repair a silver bullet for code generation? In *ICLR*, pages 1–49, Vienna, Austria. OpenReview.net.

OpenAI. 2023. GPT-4 technical report. *CoRR*, 2303.08774:1–100.

OpenAI. 2026. OpenAI GPT-5 system card. *CoRR*, 2601.03267:1–61.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, and 2 others. 2019. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8024–8035, Vancouver, BC, Canada.

Pylint-dev. 2004. Pylint. https://github.com/pylint-dev/pylint.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: memory optimizations toward training trillion parameter models. In *SC*, pages 1–16, Virtual Event / Atlanta, Georgia, USA. IEEE/ACM.

Manav Singhal, Tushar Aggarwal, Abhijeet Awasthi, Nagarajan Natarajan, and Aditya Kanade. 2024. NoFunEval: Funny how code LMs falter on requirements beyond functional correctness. In *COLM*, pages 1–25, Philadelphia, PA, USA.

Google Gemini Team. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *CoRR*, 2507.06261:1–72.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-Instruct: Aligning language models with self-generated instructions. In *ACL*, pages 13484–13508, Toronto, Canada. Association for Computational Linguistics.

Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *ESEC/FSE*, pages 172–184, San Francisco, CA, USA. ACM.

J. D. Zamfirescu-Pereira, Eunice Jun, Michael Terry, Qian Yang, and Bjoern Hartmann. 2025. Beyond code generation: LLM-supported exploration of the program design space. In *CHI*, pages 153:1–153:17, Yokohama, Japan. ACM.

Huan Zhang, Wei Cheng, Yuhan Wu, and Wei Hu. 2024. A pair programming framework for code generation via multi-plan exploration and feedback-driven refinement. In *ASE*, pages 1319–1331, Sacramento, CA, USA. ACM.

Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022. CoditT5: Pretraining for source code and natural language editing. In *ASE*, pages 22:1–22:12, Rochester, MI, USA. ACM.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. In *ACL*, pages 769–787, Toronto, Canada. Association for Computational Linguistics.

Zekai Zhang, Mingwei Liu, Zhenxi Chen, Linxi Liang, Yuxuan Chen, Guangsheng Ou, Yanlin Wang, Dan Li, Xin Peng, and Zibin Zheng. 2025. Generating high-quality datasets for code editing via open-source language models. *CoRR*, 2509.25203:1–23.

A Styles of Diff Formats

We provide an investigation into three hunk styles for content-addressed diff formats, including:

- the hunk rewrite style, which directly specifies the target code block to be rewritten. An example is shown in Figure 8.
- the unified diff-like style, which incorporates context lines alongside addition and deletion markers. An example is shown in Figure 1c.
- the search/replace style (Gauthier, 2023a), which utilizes specific delimiters to separate the original block from its replacement. An example is shown in Figure 9.

Our experimental results in Table 5 indicate that the unified diff-like style suffers from a significant decline in edit accuracy. This performance drop occurs since the inclusion of unchanged context lines often disorients the model compared to the simpler binary logic of addition and deletion. Furthermore, while the search/replace style and the hunk rewrite style demonstrate comparable accuracy, the former introduces a critical reliability risk. Specifically, the search/replace style can fail to patch if the source code contains characters identical to its special delimiters. This potential for catastrophic failure constitutes a direct violation of the reliability requirements established in Equation 1. Consequently, we adopt the hunk rewrite style for all content-addressed diff formats to ensure maximum robustness and consistency across diverse scenarios.

```
@@ ... @@
-            print("Warning.")
-            process_user(user)
-            continue
+            print("Warning.")
+            process_user(user, False)
+            continue
```

Figure 8: An example of MINCONTENTDIFF using the hunk rewrite style.

```
<<<<<<< SEARCH
            print("Warning.")
            process_user(user)
            continue
=======
            print("Warning.")
            process_user(user, False)
            continue
>>>>>>> REPLACE
```

Figure 9: An example of MINCONTENTDIFF using the search/replace style.

B Details of Experiment Setup

B.1 Details of Datasets

We include three training datasets:

- **OCEData** (Zhang et al., 2025), a high-quality, synthetic Python dataset for code editing. Its samples cover a wider range of difficulty levels and features descriptive and lazy instruction styles. We use the full set for enough samples to adapt edit formats.
- **InstructCoder** (Li et al., 2024), a synthetic Python dataset based on the Self-Instruct framework (Wang et al., 2023). It consists mainly of short, function-level snippets for general-purpose code editing.
- **CommitPackFT** (Muennighoff et al., 2024), a real-world multilingual dataset filtered from GitHub commits. However, its generally limited quality and the lack of informative commit messages yields limited improvements in model training (Aggarwal et al., 2025). We chose the JavaScript subset since it has the most samples among mainstream programming languages other than Python.

To ensure data quality, we exclude samples whose output code contains syntax errors or whose edits yield no code changes. For Python datasets, we further format both the source and target code with Black (Foundation, 2018) to minimize the impact of non-semantic modifications.

We evaluate on four diverse Python code editing benchmarks. The first two instruction-guided benchmarks aligns with the training task:

- **EditEval** (Li et al., 2024) is curated from GitHub commit data, MBPP (Austin et al., 2021), and HumanEval (Chen et al., 2021).
- **CanItEdit** (Cassano et al., 2024) is a hand-crafted benchmark featuring both descriptive and lazy editing instructions. Its samples frequently involve edits on longer code snippets.

The other two benchmarks involve bug-fixing tasks that do not provide typical instructions:

- **HumanEvalFix** (Muennighoff et al., 2024) is the Python subset of code repair scenario from HumanEvalPack, which requires models to fix a buggy function based on correct unit tests, without explicit edit instruction.

Formats	EditEval	CanItEdit	HumanEvalFix	Aider-1	Aider-2	Average
MINCONTENTDIFF	**61.42**	**37.02**	**52.20**	35.56	42.22	45.68
w/ interlaced	58.66	32.12	47.16	28.89	40.00	41.37
w/ search/replace	60.26	36.57	51.86	**37.78**	**42.96**	**45.89**
CONTENTDIFF	67.91	47.19	65.91	**43.70**	47.41	54.43
w/ interlaced	56.11	31.57	44.63	**43.70**	49.63	45.13
w/ search/replace	**69.36**	**48.38**	**66.95**	42.22	**52.59**	**55.90**
BLOCKDIFF	69.95	**48.07**	**64.85**	**45.19**	51.85	**55.98**
w/ interlaced	60.26	40.62	51.49	43.70	48.89	48.99
w/ search/replace	**70.15**	47.31	63.14	44.44	**53.33**	55.68
FUNCDIFF	**70.88**	**50.31**	**67.62**	**45.93**	51.85	**57.32**
w/ interlaced	61.01	40.31	52.13	40.00	45.19	47.73
w/ search/replace	69.87	49.05	66.04	45.19	**53.33**	56.69

Table 5: Style comparison of content-addressed diff formats on Qwen2.5-Coder-7B.

- **Aider** (Gauthier, 2023b) simulates realistic Python coding exercises involving up to two attempts. The first stage, **Aider-1**, is akin to code generation, where the model completes a starting Python file based on a rich problem description. If the associated test suite fails, the second stage, **Aider-2**, asks the model to fix the previously generated buggy code based on the `pytest` (Krekel et al., 2004) results.

The statistics of training datasets and evaluation benchmarks are presented in Table 6. Moreover, we report the proportion of the samples where the diff format is selected by ADAEDIT in Table 7.

B.2 Details of Implementation Details

Training and Inference All models are fine-tuned for 3 epochs using the AdamW optimizer with a cosine scheduler (Loshchilov and Hutter, 2019), which has a peak learning rate of 3e-5 and a warmup ratio of 0.1. Full-parameter SFT is conducted on 8 A100 (80GB) GPUs using a combination of Fully Sharded Data Parallel (FSDP2) (Paszke et al., 2019; Rajbhandari et al., 2020) and sequence parallelism (Li et al., 2023b), achieving an effective batch size of 256 with a maximum sequence length of 4,096 tokens.

For inference, we employ the vLLM library (Kwon et al., 2023) and 4 A100 GPUs for efficient inference, with a maximum generation length of 4,096 tokens. We report the results of checkpoints with the best average accuracy. Please refer to Appendix B.3 for detailed inference hyperparameters.

Prompt Template As shown in the text box, all our experiments employ a unified prompt template. During Training, the prompt ends exactly at '`### Response\n`'. The subsequent code or diff, which follows immediately, is used as the target sequence for calculating the cross-entropy loss. During Inference, to guide the model's output format, we populate the prefix that acts as a starting sequence for generation:

- For full-code generation, it is set to '` ```python\n `'.
- For any fixed diff format, it is set to '` ```diff\n `'.
- For ADAEDIT, we provide only '` ``` '. This minimal prefix intentionally leaves the format ambiguous, compelling the model to decide whether to use a diff format or full code by first producing the format specifier itself, followed by the content.

Prompt Template for Code Editing

````
### Instruction
{INSTRUCTION}

### Input Code
```python
{INPUT_CODE}
```

### Response
```python/diff
{RESPONSE}
```
````

For benchmarks (Li et al., 2024; Cassano et al., 2024) that provide explicit natural language instructions, we directly apply the above prompt template. For benchmarks (Muennighoff et al., 2024; Gauthier, 2023b) that simulate bug-fixing scenarios

Features	OCEData	InstructCoder	CommitPackFT-JavaScript	HumanEvalFix		EditEval	CanItEdit	Aider-1
				Python	JavaScript			
# Samples	59,091	101,715	52,142	164		194	210	135
Avg. # instruction	104.86	20.20	8.59	220.52	220.63	20.50	52.71	534.73
Avg. # input code	206.73	108.07	193.63	63.85	100.98	72.02	317.76	56.51
Avg. # reference	301.23	151.01	214.38	64.51	102.30	87.75	381.70	242.67
Avg. # diff lines	23.62	13.10	9.13	2.21	2.40	8.68	13.82	34.76
Max. # instruction	472	112	134	1,217	1,309	73	282	3,414
Max. # input code	882	729	663	248	432	235	1,544	2,133
Max. # reference	2,066	1,394	759	247	429	345	1,587	2,063
Max. # diff lines	322	211	127	8	10	46	59	364

Table 6: Statistics of the training dataset and evaluation benchmarks, calculated by the Qwen2.5-Coder tokenizer.

Formats	OCEData	InstructCoder	CommitPackFT-JavaScript
ContentDiff	22.35	14.78	52.54
BlockDiff	24.07	21.79	31.58
FuncDiff	19.33	16.18	27.45

Table 7: Proportion of the samples (%) where the diff format is selected by AdaEdit, using the Qwen2.5-Coder tokenizer.

````
### Instruction
Based on the correct tests, fix bugs in `has_close_elements`.

```python
def check(has_close_elements):
    assert has_close_elements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True
    ...

check(has_close_elements)
```

### Input Code
```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    ...
```

### Response
```python/diff
````

Figure 10: A prompt example of HumanEvalFix.

without explicit instructions, we adopt a standardized approach to formulate the prompt:

- **HumanEvalFix:** the instruction is generated from a fixed template with the correct unit tests. An example is provided in Figure 10.
- **Aider-2:** we use a fixed instruction with the error messages from `pytest`. An example is shown in Figure 11.

B.3 Details of Evaluation Metrics

The details of each metric are as follows:

````
### Instruction
Fix the code to resolve the testing errors.

```text
{pytest error messages ... }
 def test_underscore_emphasis(self):
>       self.assertEqual(abbreviate("The Road _Not_ Taken"), "TRNT")
E       AssertionError: 'TR_T' != 'TRNT'
{pytest error messages ... }
```

### Input Code
```python
import re

def abbreviate(words):
    ...
```

### Response
``` python/diff
````

Figure 11: A prompt example of Aider-2.

- **Accuracy:** we report the unbiased pass@1 score (Chen et al., 2021; Holtzman et al., 2020), using 20 samples with a temperature of 0.2 and a top-p value of 0.95.
- **Usability:** we report the exact patch-apply success rate, and the percentage of edited code without lint errors, using the popular Python linter `pylint` (Pylint-dev, 2004).
- **Latency:** we adapt the concept of First Meaningful Paint (Google, 2017) from web performance to IDE latency. We measure it as the average number of tokens required to generate the first output that can be meaningfully rendered to the user. For full-code generation, the code is only renderable upon completion of the entire output. For diff formats, this corresponds to the completion of the first diff hunk, which is the first moment a user can perceive a tangible change.

Base models	Formats	EditEval	CanItEdit	HumanEvalFix	Aider-1	Aider-2	Average
DeepSeek-Coder-6.7B	FULLCODE	64.38	44.88	56.95	45.19	49.63	52.21
	MINUNIDIFF	11.13	5.38	5.88	5.93	8.89	7.44
	w/ numbers	33.94	17.67	32.53	21.48	26.67	26.46
	UNIDIFF	37.89	9.10	26.80	28.15	30.37	26.46
	w/ numbers	36.03	18.33	31.28	29.63	34.07	29.87
	MINCONTENTDIFF	53.51	27.21	42.13	37.04	40.00	39.98
Qwen2.5-Coder-14B	FULLCODE	69.59	60.95	70.40	54.07	64.44	63.89
	MINUNIDIFF	29.10	11.62	19.15	24.44	31.11	23.08
	w/ numbers	52.06	34.57	51.65	31.11	43.70	42.62
	UNIDIFF	53.22	21.50	50.58	37.78	46.67	41.95
	w/ numbers	59.33	39.76	53.26	40.00	50.37	48.54
	MINCONTENTDIFF	61.75	46.93	59.73	48.15	55.56	54.42

Table 8: More results of conventional diff formats, trained on DeepSeek-Coder-6.7B and Qwen2.5-Coder-14B.

- **Cost:** we report the average number of tokens in the model's complete output, which corresponds to the overall computational cost of generating the edit.

For Aider, which involves multi-turn interactions, we report pass@1 using greedy sampling and omit efficiency comparisons for Aider-2 to ensure fairness.

For ADAEDIT, we report the percentage of samples where the model correctly selects the most token-efficient edit format. To evaluate this selection accuracy, we convert each generated output of ADAEDIT into its alternative representation to facilitate a direct comparison. By calculating and comparing their token counts, we determine whether the model successfully identified the representation with the lower cost.

C More Results of Conventional Diff Formats

As a supplement to Tables 1 and 2, we also evaluate conventional diff formats on DeepSeek-Coder-6.7B and Qwen2.5-Coder-14B. The results in Table 8 show that all number-indexed diff formats and MINCONTENTDIFF yield edit accuracies far below that of the full-code baselines, which is consistent with conclusion on Qwen2.5-Coder-7B.

D More Results on Edit Efficiency

The additional evaluations of edit latency on DeepSeek-Coder-6.7B and Qwen2.5-Coder-14B are shown in Figures 12 and 13, respectively. Moreover, the additional evaluations of edit cost are shown in Figures 14 and 15, respectively. The results also demonstrate that ADAEDIT intelligently

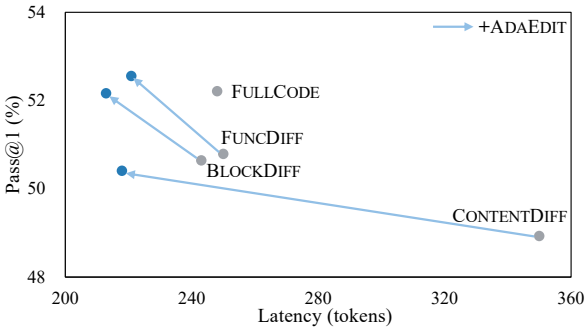

Figure 12: Latency-accuracy landscape of edit formats and ADAEDIT, trained on DeepSeek-Coder-6.7B.

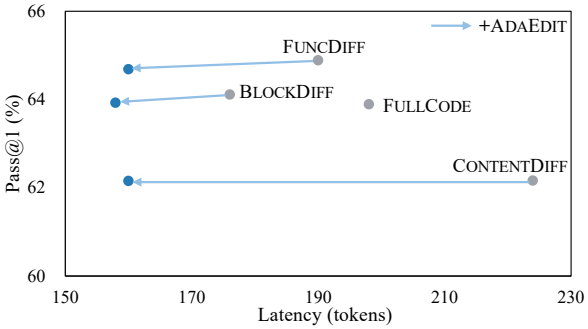

Figure 13: Latency-accuracy landscape of edit formats and ADAEDIT, trained on Qwen2.5-Coder-14B.

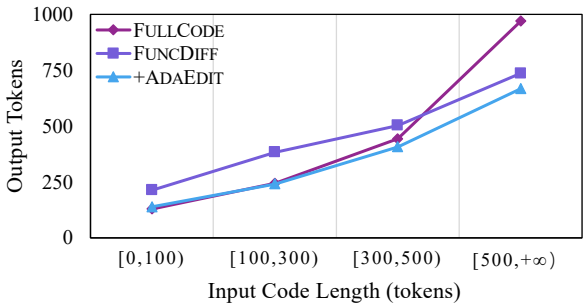

Figure 14: Edit cost comparison across code scales, trained on DeepSeek-Coder-6.7B.

Formats	EditEval	CanItEdit	HumanEvalFix	Aider-1	Aider-2	Average
Base model	55.39	42.98	**65.12**	35.56	44.44	48.70
FULLCODE	62.53	**47.31**	61.89	39.26	48.15	51.83
MINUNIDIFF	34.25	10.33	14.36	17.78	20.00	19.34
w/ numbers	48.66	23.62	47.93	24.44	30.37	35.00
UNIDIFF	45.26	13.00	27.20	31.85	34.07	30.28
w/ numbers	51.29	25.12	44.57	32.59	40.74	38.86
MINCONTENTDIFF	54.12	32.60	53.57	31.11	35.56	41.39
CONTENTDIFF	59.56	39.29	55.88	**41.48**	47.41	48.72
w/ ADAEDIT	62.16	35.05	62.90	40.74	49.63	50.10
BLOCKDIFF	62.42	42.43	63.84	40.00	49.63	51.66
w/ ADAEDIT	63.79	43.57	61.89	40.74	49.63	51.92
FUNCDIFF	**65.57**	45.05	64.42	38.52	48.89	52.49
w/ ADAEDIT	63.04	45.93	62.13	**41.48**	**50.37**	**52.59**

Table 9: Pass@1 comparison on the InstructCoder dataset, trained on Qwen2.5-Coder-7B.

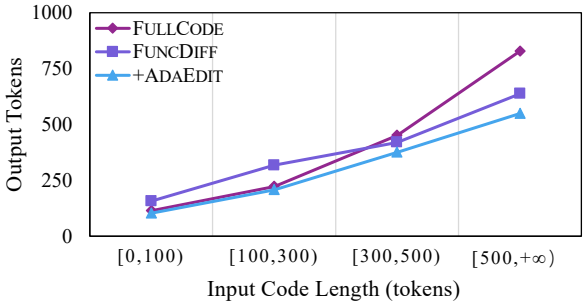

Figure 15: Edit cost comparison across code scales, trained on Qwen2.5-Coder-14B.

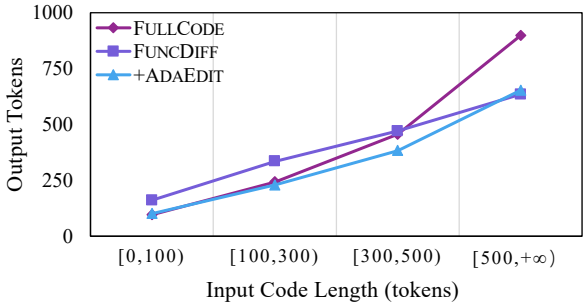

Figure 16: Edit cost comparison across code scales, trained on InstructCoder with Qwen2.5-Coder-7B.

switches between both formats, which not only reduces overall edit latency but also achieves optimal efficiency across all code scales. These show the consistent conclusions with Qwen2.5-Coder-7B.

E Inference-only Evaluation on Larger Models

While 7B-scale models are highly demanded for real-time coding tasks due to their inference efficiency, evaluating the capabilities of larger models is crucial from a research perspective to understand the broader applicability of ADAEDIT. To this end, we evaluate DeepSeek-V3.2 (DeepSeek-AI, 2025), Qwen2.5-72B-Instruct (Hui et al., 2024), and GPT-5-2025-08-07 (OpenAI, 2026) in a few-shot, inference-only setting. We design a system prompt instructing the models to choose the most token-efficient format between the full-code and BLOCKDIFF formats based on the edit task. It also includes a detailed description of the BLOCKDIFF format alongside a single demonstration.

The format selection accuracy and bias distribution of these models are presented in Table 10. The results clearly indicate that all evaluated larger models perform suboptimally in selecting the most token-efficient formats. Even the highly capable GPT-5 achieves only a 56.51% correctness rate, exhibiting a noticeable bias towards inappropriate format selection. Furthermore, DeepSeek-V3.2 and Qwen2.5-72B-Instruct exhibit extremely high "No change" rates. This phenomenon occurs since they struggle to learn the precise BLOCKDIFF format from a single example and frequently default to generating standard unified diffs, which inevitably leads to application failures during the patching process. By comparison, ADAEDIT using Qwen2.5-Coder-7B achieves a remarkable 91.02% accuracy rate. These findings strongly demonstrate that LLMs do not inherently possess the cost-benefit logic required for efficient format selection, further underscoring the necessity of ADAEDIT.

F More Results on InstructCoder

To validate the generalizability of our structure-aware diff formats and ADAEDIT, we conduct additional experiments using the InstructCoder dataset

Models	Correct	Bias $\leq 20\%$	Bias $\leq 50\%$	Bias $> 50\%$	No change
DeepSeek-V3.2	25.34	4.97	4.81	7.03	57.85
Qwen2.5-72B-Instruct	50.41	1.65	2.82	4.81	40.18
GPT-5	56.51	4.77	7.37	24.20	7.04
ADAEDIT	**91.02**	3.07	2.73	2.50	0.67

Table 10: The accuracy of the format selection mechanism using BLOCKDIFF. ADAEDIT is finetued on Qwen2.5-Coder-7B, and other LLMs are evaluated in a few-shot, inference-only setting.

with Qwen2.5-Coder-7B.

The results of edit accuracy are shown in Table 9. Evaluation results on latency and cost are illustrated in Figures 7 and 16, respectively. Although variations in the training set influence the absolute values of the evaluation metrics, the relative performance and comparative trends among different edit formats remain consistent with the conclusions drawn from OCEData.

One outlier is the results on HumanEvalFix, where the base model outperforms all fine-tuned models. We attribute the performance degradation to task shift on specific training data. All edit formats are fine-tuned on edits driven by explicit natural language instructions (e.g., "Replace function X with Y"). In contrast, samples in HumanEvalFix requires bug-fixing based on implicit information from unit tests (See Figure 10). This shift favors the original capabilities of the base model, while the specialized, instruction-tuned models exhibit a slight performance degradation on this out-of-domain task. This is a common phenomenon in fine-tuning, called as catastrophic forgetting (McCloskey and Cohen, 1989).

Beyond specific task shifts, adapting LLMs to novel edit formats relies heavily on the scale and diversity of SFT data. However, the community faces a systemic scarcity of high-quality, verifiable training corpora. Existing benchmarks are predominantly constrained to Python function-level snippets, lacking coverage for other languages and complex repository-level scenarios (Ma et al., 2025). Therefore, constructing large-scale, diverse datasets and exploring accuracy-aware adaptive strategies stand as critical directions for future research (Chen et al., 2024a; Luo et al., 2025).

G Time on Diff Generation and Patching

The generation of our structure-aware diff formats is built upon MINUNIDIFF, integrated with additional AST parsing and operations to ensure structural integrity. When combined with ADAEDIT, this process also includes the time required for text tokenization to facilitate format selection.

Operations	Avg. time	Max. time	User impact
Diff generation	3.40e-3	0.16	-
Patching	7.57e-5	7.19e-4	Negligible

Table 11: Time spent on diff generation and patching in seconds, using BLOCKDIFF combined with ADAEDIT on the OCEData dataset.

As shown in Table 11, we record the time spent on diff generation and patching on the OCEData dataset, using BLOCKDIFF combined with ADAEDIT. Since diff generation is primarily conducted during the training data preparation phase, the computational overhead is highly efficient for large-scale data processing and can be readily parallelized across multiple CPUs. In contrast, the patching process during inference involves only straightforward text search and replacement operations. As evidenced by our timing analysis, the user-perceived patching latency is nearly instantaneous and does not negatively impact the overall user experience or system responsiveness.

Furthermore, we conduct a stress test on the diff generation process. We process a massive Python file exceeding 10,000 lines, featuring complex class structures and scattered code modifications. Calculating the BLOCKDIFF for it takes merely 0.3 seconds, with the underlying block tree construction accounting for 0.28 seconds. These results demonstrate that the efficiency of structure-aware diff formats is highly robust, ensuring that computational overhead will not become a bottleneck for advanced inference techniques such as test-time scaling or future reinforcement learning pipelines.