EFFICIENTEDIT: Accelerating Code Editing via Edit-Oriented Speculative Decoding

Peiding Wang[1], Li Zhang[1], Fang Liu[1*], Yinghao Zhu[1], Wang Xu[3], Lin Shi[2*]
Xiaoli Lian[1], Minxiao Li[1], Bo Shen[4], An Fu[4]
[1]State Key Laboratory of Complex & Critical Software Environment, School of Computer Science and Engineering, Beihang University, China
[2]School of Software, Beihang University, China
[3]Tsinghua University, China
[4]Huawei Cloud Computing Technologies Co., Ltd., China
Email: {wangpeiding, fangliu, shilin}@buaa.edu.cn

***Abstract*—Large Language Models (LLMs) have demonstrated remarkable capabilities in code editing, substantially enhancing software development productivity. However, the inherent complexity of code editing tasks forces existing approaches to rely on LLMs' autoregressive end-to-end generation, where decoding speed plays a critical role in efficiency. While inference acceleration techniques like speculative decoding are applied to improve the decoding efficiency, these methods fail to account for the unique characteristics of code editing tasks, where changes are typically localized and existing code segments are reused. To address this limitation, we propose EFFICIENTEDIT, a novel method that improves LLM-based code editing efficiency through two key mechanisms based on speculative decoding: (1) effective reuse of original code segments while identifying potential edit locations, and (2) efficient generation of edit content via high-quality drafts from edit-oriented draft models and a dynamic verification mechanism that balances quality and acceleration. Experimental results show that EFFICIENTEDIT can achieve up to 10.38× and 13.09× speedup compared to standard autoregressive decoding in CanItEdit and CodeIF-Bench, respectively, outperforming state-of-the-art inference acceleration approaches by up to 90.6%. The code and data are available at https://github.com/zhu-zhu-ding/EfficientEdit.**

***Index Terms*—Code Editing, Large Language Models, Efficient Inference, Speculative Decoding**

I. INTRODUCTION

In recent years, Large Language Models (LLMs) (e.g., GPT-4 [1], Claude [2], DeepSeek-Coder [3], Qwen-Coder [4], CodeLlama [5]) have demonstrated remarkable success in code editing related tasks [6]–[8]. Concurrently, AI-powered development tools such as GitHub Copilot [9] and Cursor [10] are significantly transforming modern software development workflows by applying LLMs to code editing. Empirical studies also reveal that code editing—which includes modifying [11], [12], refactoring [13], or debugging existing code [14]—constitutes a highly frequent activity during the lifecycle of a software project [15]–[17]. Beyond the correctness of code edits, generation efficiency represents another critical factor that affects overall development productivity.

Accurately inferring precise edit locations from natural language instructions poses a significant challenge in code editing. As illustrated in Figure 1, the instruction "*Please add input validation to check if input is a list and modify the return value to include both sum and average*" specifies what to change but not necessarily where to insert new code (*e.g.*, the exact line for validation checks) or how to integrate changes with existing code structures (*e.g.*, restructuring the return statement). This ambiguity often necessitates complex reasoning from the LLM. Consequently, most existing code editing approaches employ end-to-end generation paradigm using LLMs [6]–[8], [18]–[21], which predominantly rely on autoregressive decoding to produce outputs token by token. This paradigm significantly limits the efficiency of LLM-based code editing. For instance, editing a 1024-token code snippet with DeepSeek-Coder-33B [3] requires approximately 86 seconds on 4 NVIDIA GeForce RTX 4090 GPUs.

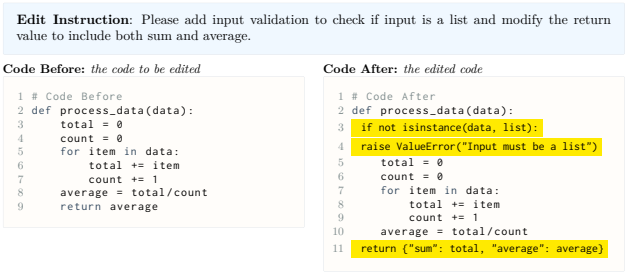


Fig. 1: A motivating example of code editing. Highlighted contents in "Code After" indicate newly generated or modified code. A significant portion of "Code After" (approximately 70%) is reused from "Code Before".

To improve LLM's inference efficiency, speculative decoding [22], [23] offers a promising solution without modifying the underlying LLM architecture. This approach employs a small draft model to rapidly generate candidate output tokens (drafts), which are then verified by the target LLM in a single forward pass to ensure drafts strictly matching the target LLM's output. Building upon the speculative decoding paradigm, various approaches are proposed to accelerate general-purpose text generation tasks [24]–[26]. Recent work has further explored replacing parameterized draft models with

* Corresponding author

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00215

retrieval systems to simplify draft model selection [27]. In particular, FastFixer [28] first applied speculative decoding to APR (Automatic Program Repair) tasks, substantially improving program repair efficiency. Most existing inference acceleration methods achieve $2 \sim 3\times$ speedup in inference, effectively improving LLMs' inference speed.

Although these efforts have shown promising performance in accelerating LLM inference, we have identified the following problems in code editing:

- **Redundant Code Generation.** End-to-end code editing with LLMs often produces redundant code due to the autoregressive decoding mechanism [29], [30], leading to inefficient inference and unnecessary computational overhead. As shown in Figure 1, the generated output contains approximately 70% redundant code segments.
- **Limitations of Speculative Decoding.** Limited by the ordinary quality of the draft model and the strict verification process, the benefits of using speculative decoding in code editing scenarios are not significant. It still takes approximately 30-40 seconds to generate an edited code of 1024 tokens for DeepSeek-Coder-33B when running on 4 NVIDIA GeForce RTX 4090 GPUs.

To address these limitations, we propose EFFICIENTEDIT, an efficient edit approach for code editing via speculative decoding. As indicated in Figure 1, to leverage the abundant redundant fragments in the code to be edited, we employ speculative decoding to efficiently reuse these fragments while identifying potential edit locations through a single forward inference of the target model. Once potential edit positions are determined, we enhance the target model's editing efficiency using high-quality drafts generated by a draft model fine-tuned with our proposed edit-content-oriented strategy. Moreover, we introduce an effective verification mechanism that optimizes the trade-off between generation quality and efficiency, further improving the target model's performance in generating edit content. Extensive evaluation across diverse benchmarks, including crowd-sourced edits (CanItEdit [6]) and interactive editing (CodeIF-Bench [31]) spanning function-, class-, and repository-level tasks, demonstrates substantial inference speedup, achieving up to 8.26× for Qwen-32B-Instruct and 13.09× for DeepSeek-Coder-33B-Instruct, significantly outperforming baseline acceleration methods.

Our key contributions are summarized as follows:

- We propose EFFICIENTEDIT, an efficient approach for improving LLM's code editing efficiency by synergistically combining code reuse with accelerated generation of new content.
- We propose the edit-oriented draft models that improves the target LLMs' code editing efficiency in speculative decoding.
- We conduct a comprehensive evaluation of EFFICIENTEDIT and demonstrate that it achieves state-of-the-art code editing efficiency, while maintaining or even improving edit quality.

II. PRELIMINARIES

A. Autoregressive Decoding

Autoregressive decoding is a prevalent approach in LLM inference, wherein output tokens are sequentially generated through progressive forward predictions.

Formally, given the LLM $\texttt{LLM}$ and the prefix token sequence $(t_1, ..., t_{n-1})$ and a new input token t_n, the model predicts the probability distribution of the subsequent token as follows:

$$\boldsymbol{p}_{n+1} = \texttt{LLM}(t_n; t_1, ..., t_{n-1}) \tag{1}$$

The newly generated token t_{n+1} is either selected as the top-1 prediction from $\boldsymbol{p}_{n+1}$ or probabilistically sampled according to the distribution $\boldsymbol{p}_{n+1}$. Then the new selected token t_{n+1} is input to the LLM to generate the next token.

Inherently, the token-by-token generation mechanism of autoregressive decoding introduces computational inefficiencies, leading to a linear escalation of inference latency that is proportional to both the generated sequence length and the model's architectural complexity.

B. Speculative Decoding

To enhance decoding efficiency, Speculative decoding has been proposed in recent literature [22], [23]. This innovative approach accelerates autoregressive generation through a draft-and-verify paradigm. During the draft stage, these methods employ a fast-inference draft model ($\texttt{LLM}_{\text{draft}}$) to generate γ candidate tokens (drafts). These candidate tokens are subsequently verified in parallel via a forward inference through the target Large Language Model ($\texttt{LLM}_{\text{target}}$), thereby significantly improving inference speed.

During the draft stage, a draft model generates M draft tokens $(d_1, \cdots, d_M)$ autoregressively given the prefix token sequence $(t_1, t_2, \cdots, t_n)$.

$$\boldsymbol{q}_{M+1} = \texttt{LLM}_{\text{draft}}(d_M; t_1, ..., t_n, d_1, \cdots, d_{M-1}) \tag{2}$$

Token d_{M+1} is sampled from the distribution of $\boldsymbol{q}_{M+1}$ (but usually greedy).

During the verify stage, the M draft tokens $(d_1, \cdots, d_M)$ are input to the target language model in a forward pass as follows:

$$\boldsymbol{p}_1, \cdots, \boldsymbol{p}_M = \texttt{LLM}_{\text{target}}(d_1, \cdots, d_M; t_1, ..., t_n) \tag{3}$$

Speculative decoding is used to determine which token would be accepted as follows:

$$\epsilon_i < \frac{\boldsymbol{p}_i[d_i]}{\boldsymbol{q}_i[d_i]} \tag{4}$$

where $\epsilon_i \sim \mathcal{U}([0, 1])$. $\boldsymbol{p}_i[d_i]$ and $\boldsymbol{q}_i[d_i]$represents the token d_i corresponding probability in $\boldsymbol{q}_i$ and $\boldsymbol{p}_i$. ϵ_i is a given hyperparameter. Intuitively, the draft token can be accepted if it has a higher probability in the target model's distribution. For lower-probability draft tokens, the decision is made stochastically based on the probability difference.

When the $\texttt{LLM}_{\text{target}}$ rejects the draft token d_i, the above process is repeated by resampling a token t_i from the target

model's probability vector $\boldsymbol{p}_i$. In the ideal case, all M tokens generated by $\texttt{LLM}_{\text{draft}}$ are accepted by $\texttt{LLM}_{\text{target}}$ through speculative decoding. Moreover, a new token t_{M+1} would be sampled from the probability distribution $\boldsymbol{p}_M$. Thus $M+1$ tokens are accepted by a single forward of the target model, thereby significantly improving its inference speed.

C. Task Definition

This study investigates end-to-end code editing tasks where edit positions cannot always be accurately determined from natural language specifications alone. We formulate the code editing task as an end-to-end sequence transformation problem:

$$c' = E(c, I) \tag{5}$$

where c is the original input code to be modified, I denotes the edit instruction (requirement), and c' is the output code after applying the edit. Each line of the output code $l_i \in c'$ can be further categorized into two subsets:

- Reused lines from the input code: $\mathcal{R} = \{l_i | l_i \in c \cap c'\}$
- Newly generated lines: $\mathcal{G} = \{l_i \in c' \setminus c\}$

Thus, the output code c' is formally represented as:

$$c' = \{l_1, l_2, \ldots, l_n\}, \text{ where } l_i \in (\mathcal{R} \cup \mathcal{G}) \tag{6}$$

III. Methodology

In this section, we present our proposed approach, EFFICIENTEDIT, for enhancing LLM-based code editing efficiency through speculative decoding. As shown in Figure 2, EFFICIENTEDIT operates in two key phases *Reuse* and *Edit*. *Reuse* treats the original code (or the remaining code) as the high-quality draft and performs parallel verification via a single forward pass of the target LLM, enabling efficient reuse of unmodified code segments. The draft segments rejected by the target LLM are preserved as candidate drafts for subsequent processing. Then, *Edit* uses the rejection points identified by the target LLM as starting positions for generating edited content. This process is further optimized through an edit-oriented draft model and a novel entropy-aware verification mechanism. After completing the edited content, a prefix matching algorithm is used to continue reusing code segments from the candidate drafts. This iterative reuse-generate paradigm significantly accelerates LLM-based code editing while maintaining output quality.

A. Reuse: Reuse Code from Code to be Edit

EFFICIENTEDIT leverages c as high-quality drafts for accelerated reuse of $\mathcal{R}$ and identifies editing positions for newly generated code $\mathcal{G}$. Specifically, since there is usually a large amount of redundant code segments in the code to be edited c and the edited code c', we employ speculative decoding to parallelize and reuse these repetitive code segments via the $\texttt{LLM}_{\text{target}}$. For speculative decoding, the selection of γ is import and challenge. While a larger γ can reduce the number of costly $\texttt{LLM}_{\text{target}}$ inferences when draft quality is high, excessive candidate token rejections may conversely increase verification overhead and draft generation time.

In code editing, we exploit the structural properties of code: c' often reuses complete lines or segments from c. Furthermore, c, as a natural draft, does not require any additional generation time. This observation motivates our approach of directly using c (or its remaining portions) as the draft to reuse these multi-token code segments and significantly reduce inference costs. This approach achieves a key advantage: **there is no need to determine an optimal draft length γ for reuse, as a single forward pass by $\texttt{LLM}_{\text{target}}$ can verify a long sequence of tokens from c.** After $\texttt{LLM}_{\text{target}}$ accepts reusable code segments in a single forward inference, intuitively, token rejection by $\texttt{LLM}_{\text{target}}$ implicitly identifies the potential starting position of a segment in $\mathcal{G}$, effectively pinpointing an edit location without additional computational cost. At this point, EFFICIENTEDIT transitions to the *Generate* phase to efficiently generate $\mathcal{G}$. When in the *Generate* phase and needing to switch back to *Reuse*, we employ a prefix matching algorithm to identify prefixes of the remaining c that match the suffix of the currently generated $\mathcal{G}$, using these match points as new draft starting points. The subsequent tokens from these positions in c then serve as draft candidates for parallel verification by $\texttt{LLM}_{\text{target}}$. Another key advantage of *Reuse* is that **it reuses code while also achieving self-editing positioning.**

Given the current generated prefix $prefix$ and the original code c (or its unverified remainder), the *Reuse* phase uses a portion of c as a draft sequence $draft = (d_1, d_2, \ldots, d_m)$. This $draft$ is appended to $prefix$. A single forward pass of $\texttt{LLM}_{\text{target}}$ over $(prefix, draft)$ yields a sequence of next-token probability distributions $(\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_m)$, where $\boldsymbol{p}_i$ is the distribution for the token in $draft$. Now, the verification method is as follows:

$$\text{decode}(\boldsymbol{p}_i) = d_i \quad \forall i \leq j \tag{7}$$

where the accepted length of $draft$ is the maximum of j satisfied above equation. Effectively, tokens from $draft$ are accepted as long as they match the greedy decoding output of $\texttt{LLM}_{\text{target}}$ at each position. If a mismatch occurs at token d_j, all preceding accepted tokens $(d_1, \ldots, d_j)$ extend $prefix$ and switch to the *Generate*. At the same time, we treat the remaining drafts $(d_{j+1}, \ldots, d_m)$ as reusable candidate drafts. In this paper, the decode function in Equation. (7) uses greedy decoding, i.e., the token with the highest probability is selected.

B. Generate: Efficient Edit Generation

When editing large amounts of content, the sequential nature of autoregressive decoding emerges as a critical efficiency bottleneck. And the inherent flexibility and complexity of code editing tasks pose significant challenges for obtaining and reusing high-quality drafts at edit locations similar to *Reuse*. To address this, we adopt a speculative decoding paradigm that leverages both target and draft models to accelerate the generation of edited content.

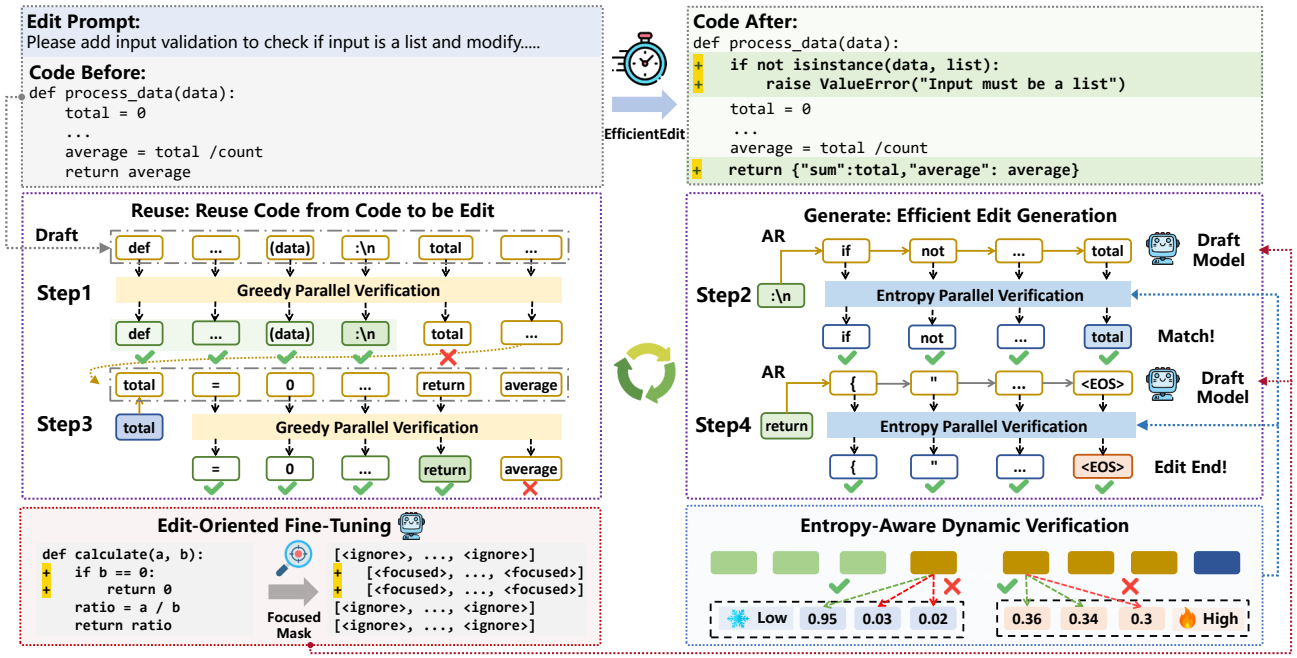


Fig. 2: The overview of EFFICIENTEDIT.

EFFICIENTEDIT efficiently generates the edit content $\mathcal{G}$ by employing an $\texttt{LLM}_{\text{draft}}$, fine-tuned via *Edit-Oriented Fine-Tuning*, to generate high-quality drafts that expedite $\texttt{LLM}_{\text{target}}$'s inference. Additionally, we introduce an *Entropy-Aware Dynamic Verification* mechanism to balance generation quality and the acceleration effect for $\texttt{LLM}_{\text{target}}$. The key insight is that **the $\texttt{LLM}_{\text{draft}}$ primarily needs to generate contextually appropriate drafts at $\texttt{LLM}_{\text{target}}$'s edit positions, and does not require strict greedy decoding-based verification across all tokens to achieve generation quality comparable to that of $\texttt{LLM}_{\text{target}}$.**

Edit-Oriented Fine-Tuning. In EFFICIENTEDIT, $\texttt{LLM}_{\text{draft}}$ must produce high-quality drafts to accelerate $\texttt{LLM}_{\text{target}}$'s generation of $\mathcal{G}$. However, despite potential distributional similarity between $\texttt{LLM}_{\text{draft}}$ and $\texttt{LLM}_{\text{target}}$, it is usually difficult for $\texttt{LLM}_{\text{draft}}$ to generate token sequences that perfectly match $\texttt{LLM}_{\text{target}}$'s output. This difficulty can increase the verification cost for $\texttt{LLM}_{\text{target}}$, thereby affecting the acceleration. To address this issue, we propose a training strategy focused on edit positions. First, we use $\texttt{LLM}_{\text{target}}$ to generate c' from I and c. Next, we identify lines in c' that do not exist in c and label them as $\mathcal{G}$. By masking all $\mathcal{R}$ tokens in c' except those belonging to $\mathcal{G}$, we exclude $\mathcal{R}$ from loss computation during the fine-tuning of $\texttt{LLM}_{\text{draft}}$. This design is motivated by the intuition that $\texttt{LLM}_{\text{draft}}$ does not need to learn how to perfectly generate $\mathcal{R}$ or locate the editing position for $\texttt{LLM}_{\text{target}}$; instead, it should focus solely on accurately predicting $\mathcal{G}$. Based on this, we design the following loss-masking training strategy:

$$\mathcal{L}_{\text{EoFT}} = \sum_{i \in \mathcal{G}_{\text{tokens}}} \mathcal{L}_{\text{origin}}(y_i|x_i) \tag{8}$$

where $\mathcal{L}_{\text{origin}}$ is the cross-entropy loss, y_i denotes the $\texttt{LLM}_{\text{target}}$ label of the i-th token within the tokenized representation of $\mathcal{G}$ (denoted $\mathcal{G}_{\text{tokens}}$), and x_i is the corresponding token generated by $\texttt{LLM}_{\text{draft}}$.

Entropy-Aware Dynamic Verification. Another critical factor influencing the performance of draft models in enhancing the inference performance of $\texttt{LLM}_{\text{target}}$ is the validation process. On one hand, since draft models are hard to generate outputs identical to $\texttt{LLM}_{\text{target}}$, an overly strict validation mechanism reduce speedup gains and reject correct content generated by $\texttt{LLM}_{\text{draft}}$. On the other hand, relaxing the validation threshold may accept the incorrect code so compromise output quality. As shown in Figure 2, the LLM's probability distribution at each position can exhibit substantial variation. When the highest-probability token dominates the distribution with a significantly greater probability than other tokens, this reflects high confidence in the target LLM's inference; conversely, when the top-1 token's probability margin over alternatives is narrow, it indicates low confidence. Inspired by this observation, we propose an adaptive verification mechanism: for positions where the $\texttt{LLM}_{\text{target}}$ demonstrates high confidence (characterized by a large probability gap between the top token and others), the $\texttt{LLM}_{\text{draft}}$'s output must exactly match the target LLM's prediction, while for low-confidence positions, we relax the verification threshold to accept the draft output if it matches any of the target LLM's top_τ candidate tokens. Specifically, given the base threshold hyperparameter k, at each token validation step we select the first top_k probability vectors from $\texttt{LLM}_{\text{target}}$ at this token and compute their entropy H_{norm} using the following formula:

$$H_{\text{norm}} = \frac{-\sum \boldsymbol{p_i} \log \boldsymbol{p_i}}{\log top_k}, \ i \in \{1, \ldots, top_k\} \tag{9}$$

top_τ represents the threshold for draft tokens generated by the $\texttt{LLM}_{\text{draft}}$ to be accepted by the $\texttt{LLM}_{\text{target}}$. Specifically, the draft token is accepted when the highest probability token of $\texttt{LLM}_{\text{draft}}$ is in the first top_τ positions in the distribution of $\texttt{LLM}_{\text{target}}$.

$$top_\tau = \begin{cases} \lceil k \cdot H_{\text{norm}} \rceil & \text{if } \lceil k \cdot H_{\text{norm}} \rceil \geq 1 \\ 1 & \text{if } \lceil k \cdot H_{\text{norm}} \rceil < 1 \end{cases} \tag{10}$$

Higher entropy values indicate greater uncertainty in the $\texttt{LLM}_{\text{target}}$ of the current location, which leads to a larger threshold top_τ. Conversely, the lower the entropy value, the lower the uncertainty of $\texttt{LLM}_{\text{target}}$, leading to a smaller threshold top_τ. In particular, top_τ takes 1 when the threshold is less than 1, i.e., $\texttt{LLM}_{\text{draft}}$ and $\texttt{LLM}_{\text{target}}$ must be strictly consistent. This validation method balances the acceptance rate and quality of high-entropy locations by adaptively adjusting top_τ, while maintaining strict criteria for low-entropy locations.

IV. Experimental Setups

A. Benchmarks

To comprehensively evaluate EFFICIENTEDIT on diverse code editing tasks, we employ the crowd-sourced benchmark CanItEdit [6] and the dialogue-based iterative editing benchmark CodeIF-Bench [31].

- **CanItEdit** [6] contains 105 crowd-sourced Python code editing tasks, each with: (1) input/output code snippets, (2) two natural language instructions (descriptive and lazy), and (3) a hidden test suite. The dataset covers diverse domains (data structures, algorithms, NLP, etc.) and requires familiarity with Python libraries like NumPy and PyTorch. Tasks are categorized as corrective, refinement, or adaptive.
- **CodeIF-Bench** [31] evaluates interactive code generation through multi-round dialogues, each dialogue set comprises an initial programming task (first round) followed by multiple independent editing instructions. It classifies tasks into three levels: L-1 (stand-alone), L-2 (intra-file context), and L-3 (cross-file context). We focus on L-1/L-2 across 9 real-world software instruction types, spanning algorithmic and repository-level tasks. L-3 is excluded due to context length and resource constraints. In this study, we exclusively evaluate the 2-Round scenario, where the first-round output serves as input code for subsequent editing, yielding a total of 762 editing problems with test cases.

B. Training Dataset

We select **InstructCoder** [11] as the initial training dataset to obtain the high-quality draft model. InstructCoder is an instruction-tuning dataset for adapting LLMs to code editing tasks, covering repairs, refactoring, and transformations. Using self-instruct [32] on GitHub-derived seed data, it generates 110K+ high-quality (I, c, c') triples spanning diverse editing scenarios, originally generated by GPT-3.5 [33].

C. Metrics

1) Edit Capability: We use the **Pass@K** [34] metric to measure the LLMs' code edit capabilities. This metric is widely used in code generation and editing tasks [6], [8], [34]. It reflects the probability that the model successfully generates functionally correct code at least once in K attempts, computed as:

$$\text{Pass@K} = 1 - \frac{\binom{n-c}{K}}{\binom{n}{K}} \tag{11}$$

where n is the total number of generated code samples, c is the number of correct solutions among these samples, and K is the number of allowed attempts. In this paper, we primarily report Pass@1.

2) Inference Efficiency: Following existing work [24], [27], [28], we use the following metrics to measure LLM inference efficiency for code editing:

- **Tokens/s**: This metric represents the average number of tokens generated by LLMs per second during inference without mini-batching.
- **Speedup**: This metric measures the average speedup factor in tokens generated per second when using an inference acceleration method versus standard autoregressive inference.

D. Baselines

To assess the effectiveness of EFFICIENTEDIT, we compare it with the following baselines: autoregressive inference (AR), standard speculative decoding (SD) [22], the state-of-the-art method Ouroboros [24] in NLP, and FastFixer [28] in program repair.

- **Autoregressive Decoding (AR)**: AR is the fundamental token-by-token decoding approach employed by existing LLMs. In our evaluation, the autoregressive implementations of all methods use KV-Cache, which improves inference efficiency by memorizing key-value pairs in attention layers.
- **Speculative Decoding (SD)**: In this work, we adopt the standard speculative decoding framework where a smaller draft model generates candidate tokens, which are then verified in parallel by a larger target model. Note we use greedy decoding for the verification process in this baseline.
- **FastFixer**: FastFixer is a speculative decoding-based inference acceleration framework specifically designed for program repair. The method achieves substantial speedup by searching and reusing code segments as drafts from historical buggy code. We adapt the original FastFixer by replacing historical buggy code with the code that needs to be edited for general code editing tasks.
- **Ouroboros**: Ouroboros enhances draft model generation through phrases. The method employs heuristic techniques to extract reusable phrases from both rejected draft tokens and historical conversation contexts, thereby accelerating generation and improving speculative decoding performance. We adapt Ouroboros's phrase extraction approach from historical contexts to code editing, establishing it as an effective baseline.

E. Implementation Details

Models: We select the following model combinations to demonstrate the effectiveness of EFFICIENTEDIT:

- **Qwen2.5-Coder Configuration:** We employ Qwen2.5-Coder-32B-Instruct as the target model combined with Qwen2.5-Coder-7B-Base as the draft model.
- **DeepSeek-Coder Configuration:** We employ DeepSeek-Coder-33B-Instruct as the target model combined with DeepSeek-Coder-6.7B-Base as the draft model.

To avoid potential knowledge conflicts [7] with instruction-tuning data in our fine-tuning phase, we select the base models (Qwen2.5-Coder-7B-Base and DeepSeek-Coder-6.7B-Base) as the draft models and fine-tune them as the draft models in EFFICIENTEDIT.

Draft Model Training Data Preparation: We randomly select 20K samples from the InstructCoder dataset and regenerate responses for each using DeepSeek-V3 (as DeepSeek-Coder-33B-Instruct API was unavailable) and Qwen-32B-Instruct. We use a rule-based method to distinguish between edit content $\mathcal{G}$ and reused code $\mathcal{R}$ in the edited code to apply our masked loss for fine-tuning the draft models.

Training Configuration: Fine-tuning is performed using LoRA on 2 × NVIDIA RTX 4090 GPUs with hyperparameters: the maximum sequence length of 4096, the learning rate of 2e-5, the LoRA rank of 16, and the batch size of 32.

Hardware and Implementation: All experiments are performed on 5 × NVIDIA RTX 4090 GPUs, allocating 4 GPUs for the target model and 1 GPU for the draft model. The CPU used is an Intel(R) Xeon(R) Gold 6330 CPU. For the Qwen2.5-Coder configuration, the base entropy threshold k is set to 3; for the DeepSeek-Coder configuration, k is set to 5. Except for EFFICIENTEDIT's entropy-aware verification of edit content, all other speculative decoding methods (including the reuse phase of EFFICIENTEDIT) use greedy decoding for verification. The hyperparameter draft length γ for all evaluated methods need to set is 7.

V. EXPERIMENTAL RESULTS

We aim to address the following four research questions:

- **RQ1:** How does EFFICIENTEDIT perform on code editing? This question evaluates the overall effectiveness of EFFICIENTEDIT in terms of both inference efficiency and edit quality compared to baselines.
- **RQ2:** How does each component of EFFICIENTEDIT contribute to its performance? This question investigates the individual impact of EFFICIENTEDIT's core components.
- **RQ3:** How effective is Entropy-Aware Dynamic Verification compared to other verification methods? This question compares our verification mechanism against other strategies.
- **RQ4:** How does EFFICIENTEDIT perform in code editing tasks with different code reuse rates? This question examines the performance of EFFICIENTEDIT across scenarios with varying proportions of reused versus edit content.

A. RQ1: Overall Performance of EFFICIENTEDIT

To answer this research question, we compared our approach with three state-of-the-art inference acceleration baseline methods: SD, FastFixer, and Ouroboros. AR represents the autoregressive decoding results with KV-Cache of target models. Except for FastFixer, all other methods require the draft model. The results are presented in Table I. Due to implementation constraints in the original Ouroboros framework that preclude compatibility with Qwen-series models, we have not reported its results on the Qwen series models.

In terms of inference efficiency, our model surpasses all baselines, achieving highest inference acceleration of 8.26× and 13.09× on the Qwen2.5-Coder configuration and DeepSeek-Coder configuration. In addition, compared to the best baseline FastFixer, we can even improve the efficiency by up to 90.6% in Qwen2.5-Coder configuration. Among all baselines, FastFixer delivers the best performance. We attribute this to the high similarity between program repair and code editing tasks, enabling its retrieval-based approach to also achieve gains in code editing tasks. Although this method achieves improved acceleration by reusing reusable code segments without relying on draft models, its performance degrades on data with autoregressive edit generation and excessive retrieval verification overhead. Ouroboros outperforms SD by reusing the verification process and historical context phrases to improve draft generation speed, achieving overall acceleration. However, it remains limited by the computational overhead of draft generation and repeatedly generated 'redundant' code segments. In contrast, our approach dynamically and efficiently reuses reusable code from code to be edited, requiring only milliseconds of target model for a forward inference time. Furthermore, by optimizing the SD-based inference acceleration scheme at edit locations, our approach balances speed and effectiveness. As a result, EFFICIENTEDIT demonstrates superior robustness and achieves the best acceleration performance across scenarios—whether the general function-level and class-level editing tasks CanItEdit or dialogue editing tasks CodeIF-Bench.

In terms of edit quality (functional correctness of the edited code), EFFICIENTEDIT achieves an effective balance between acceleration and edit quality. We use the greedy decoding results of the target model as the quality baseline to verify whether our dynamic threshold validation affects the generation quality. In all model combinations, we maintained quality consistent with greedy decoding in most cases. This indicates that our dynamic threshold verification method can further improve speed without compromising generation quality. In addition, in some cases, it can even surpass the greedy decoding results. Specifically, Qwen2.5-Coder configuration even surpasses results of the target model using greedy decoding in quality (e.g. 55.2 vs 54.3 and 70.5 vs 68.6 on CanItEdit). Notably, the training data was annotated solely using the Qwen-Coder-32B-Instruct model without employing stronger closed-source models, yet still delivers improved results. While larger models typically exhibit superior code editing capabilities, we found that that draft models (especially the strong Qwen2.5-Coder) can also edit some data incorrectly edited by the target model alone. Our dynamic threshold validation effectively combines the strengths of both small and large models, achieving editing quality that sometimes surpasses the target model alone. This holds also for the DeepSeek-

TABLE I: The overall performance of EFFICIENTEDIT and the baselines on code editing datasets.

Backbone	Approach	CanItEdit						CodeIF-Bench					
		Lazy			Descriptive			L-1			L-2		
		Token/s	Speedup	Pass@1	Token/s	Speedup	Pass@1	Token/s	Speedup	Pass@1	Token/s	Speedup	Pass@1
Qwen2.5-Coder	AR	13.0	1.00×	54.3	13.0	1.00×	68.6	13.4	1.00×	43.1	12.7	1.00×	4.2
	SD	22.9	1.76×	54.3	23.2	1.78×	68.6	23.1	1.73×	43.1	22.6	1.78×	4.2
	FastFixer	37.1	2.85×	54.3	38.6	2.97×	68.6	60.8	4.54×	43.1	97.0	7.64×	4.2
	Ouroboros	-	-	-	-	-	-	-	-	-	-	-	-
	EFFICIENTEDIT	**70.7**↑90.6%	**5.44**×	**55.2**	**70.0**↑81.3%	**5.38**×	**70.5**	**81.4**↑33.9%	**6.07**×	43.1	**104.9**↑8.1%	**8.26**×	**4.5**
DeepSeek-Coder	AR	11.8	1.00×	49.5	11.8	1.00×	60.0	11.6	1.00×	35.8	12.0	1.00×	8.4
	SD	19.6	1.67×	49.5	19.9	1.69×	60.0	18.5	1.59×	35.8	18.7	1.56×	8.4
	FastFixer	77.9	6.60×	49.5	61.7	5.23×	60.0	93.7	8.08×	35.8	128.5	10.71×	8.4
	Ouroboros	36.1	3.06×	49.5	36.1	3.06×	60.0	31.8	2.74×	35.8	33.2	2.77×	8.4
	EFFICIENTEDIT	**122.5**↑57.3%	**10.38**×	48.5	**94.6**↑53.3%	**8.02**×	59.0	**111.7**↑19.2%	**9.63**×	**36.5**	**157.1**↑22.3%	**13.09**×	**8.7**

Coder configuration, which shows a marginal performance decrease CanItEdit because the inherent differences in editing capabilities between the target and draft model but matches greedy decoding overall and outperforms it on CodeIF-Bench. These results demonstrate our method's generalisation across model combinations and ability to maintain the acceleration-quality trade-off across diverse editing tasks.

RQ1 Summary: Our method substantially outperforms existing SOTA acceleration methods. Compared to the best-performing baseline FastFixer, EFFICIENTEDIT improves efficiency by an average of **70.6%** on CanItEdit and **21.6%** on CodeIF-Bench. Concurrently, EFFICIENTEDIT preserves—and in many cases even surpasses—the generation quality of greedy decoding.

B. RQ2: Contributions of Each Component

To assess the contribution of each component, we perform ablation studies on EFFICIENTEDIT. Specifically, we adopt the basic SD method as the baseline and incrementally incorporate our components, analyzing the performance gains at each stage. The experimental results are presented in Table II. For clarity, we decompose SD into two key aspects: draft model type and decoding method. Here, *Base* denotes the original base model serving as the draft model, *Greedy* indicates greedy decoding verification, *Reuse* represents the module that reuses code from the code to be edited, *Our* refers to the draft model fine-tuned with our edit-oriented loss (Equation. (8)), *SFT* refers to a draft model fine-tuned with standard supervised fine-tuning loss, and *Entropy* signifies our Entropy-Aware Dynamic Verification.

The experimental results demonstrate that each module of EFFICIENTEDIT contributes to performance improvement. Notably, the most substantial gains occur after integrating the *Reuse* module. This module reuses a large number of tokens from the original code through a single forward inference of the target model, thereby greatly reducing generation time. Additionally, it lightly locates potential editing positions and automatically controls whether to reuse code or efficiently generate edited content, highlighting the role of the *Reuse* module as a key component of EFFICIENTEDIT.

We present comparison results with standard fine-tuning methods (*SFT*) versus our edit-oriented fine-tuning (*Our*). Except for the training loss, all other experimental settings remain identical. The results demonstrate that traditional fine-tuning methods exhibit lower robustness. Specifically, in the Qwen2.5-Coder model ensemble, *SFT* achieves comparable performance to *Our* on the CanItEdit benchmark but underperforms on CodeIF-Bench. Conversely, in the DeepSeek-Coder model ensemble, traditional fine-tuning leads to significantly degraded generation quality on CanItEdit while outperforming EFFICIENTEDIT on CodeIF-Bench. We hypothesize that the loss functions in traditional fine-tuning lack task-specific optimization, causing the model to potentially learn to exploit redundant code generation to minimize training loss, thereby limiting generalization across benchmarks. In contrast, EFFICIENTEDIT's targeted loss training enhances performance at edit patch positions and improves overall effectiveness.

We further observe that while the fine-tuned draft model enhances acceleration, the improvements are modest if only fine-tuning is applied without other optimizations. This suggests that parameter discrepancies between the draft and target models can lead to output distribution differences that may not be fully resolved through fine-tuning alone. However, the *Entropy* verification method provides additional acceleration benefits and generally surpasses greedy decoding in performance. These results support our hypothesis: strict alignment between the draft model's output and the target model's distribution is not essential for maintaining quality if a smart verification strategy is used. The *Entropy* method effectively optimizes the trade-off between generation quality and acceleration, leading to performance enhancement.

RQ2 Summary: The *Reuse* module represents the most crucial component of EFFICIENTEDIT for speedup. *Edit-Oriented Fine-Tuning* and *Entropy-Aware Dynamic Verification* further enhance EFFICIENTEDIT's overall performance by accelerating edit generation.

C. RQ3: Comparison with Different Draft Verification Methods

To compare different draft verification methods within the *Generation* phase of EFFICIENTEDIT, we evaluate the following approaches:

- Greedy: The draft model's greedy decoding output must exactly match the target model's greedy decoding output for acceptance.

TABLE II: Ablation study results on code editing datasets.

Backbone	Approach	CanItEdit						CodeIF-Bench					
		Lazy			Descriptive			L-1			L-2		
		Token/s	Speedup	Pass@1	Token/s	Speedup	Pass@1	Token/s	Speedup	Pass@1	Token/s	Speedup	Pass@1
Qwen2.5-Coder	Base+Greedy	21.7	1.67×	54.3	22.0	1.69×	68.6	22.8	1.70×	43.1	22.5	1.77×	4.2
	Reuse+Base+Greedy	60.5	4.65×	54.3	63.5	4.88×	68.6	70.6	5.27×	43.1	88.0	6.93×	4.2
	Reuse+Our+Greedy	63.7	4.90×	54.3	65.9	5.07×	68.6	79.9	5.96×	43.1	99.8	7.86×	4.2
	Reuse+SFT+Entropy	70.4	5.42×	**56.2**	68.3	5.25×	**70.5**	75.7	5.65×	42.9	100.3	7.90×	4.2
	Reuse+Our+Entropy	**70.7**	**5.44**×	55.2	**70.0**	**5.38**×	**70.5**	**81.4**	**6.07**×	43.1	**104.9**	**8.26**×	**4.5**
DeepSeek-Coder	Base+Greedy	18.9	1.60×	49.5	19.1	1.62×	60.0	18.6	1.60×	35.8	18.2	1.52×	8.4
	Reuse+Base+Greedy	97.8	8.29×	49.5	79.3	6.72×	60.0	103.9	8.96×	35.8	138.1	11.51×	8.4
	Reuse+Our+Greedy	105.1	8.91×	49.5	85.0	7.20×	60.0	105.9	9.13×	35.8	140.2	11.68×	8.4
	Reuse+SFT+Entropy	**127.1**	**10.77**×	43.8	90.7	7.69×	54.3	**112.6**	**9.71**×	**37.0**	**157.7**	**13.14**×	**9.8**
	Reuse+Our+Entropy	122.5	10.38×	48.5	**94.6**	**8.02**×	59.0	111.7	9.63×	36.5	157.1	13.09×	8.7

TABLE III: Comparison results of different verification methods. The model combination is DeepSeek-Coder.

Approach	CanItEdit						CodeIF-Bench					
	Lazy			Descriptive			L-1			L-2		
	Token/s	Speedup	Pass@1	Token/s	Speedup	Pass@1	Token/s	Speedup	Pass@1	Token/s	Speedup	Pass@1
Greedy	105.1	8.91×	**49.5**	85.0	7.20×	**60.0**	105.9	9.13×	35.8	140.2	13.32×	8.4
Top-k=3	128.1	10.86×	44.8	102.5	8.68×	50.5	112.9	9.78×	35.6	159.8	13.31×	9.3
Top-k=5	131.4	11.14×	43.8	**102.7**	**8.70**×	48.6	113.4	9.77×	35.4	160.8	13.43×	**9.8**
Direct	**131.9**	**11.18**×	43.8	102.0	8.64×	47.6	**114.4**	**9.86**×	35.1	**161.2**	**13.89**×	**9.8**
Entropy	122.5	10.38×	48.5	94.6	8.02×	59.0	111.7	9.63×	**36.5**	157.1	13.09×	8.7

- Direct: The draft model's greedy decoding outputs are used directly without any verification by the target model.
- Top-k: The draft model's greedy decoding output is accepted if it appears among the top-k tokens (ranked by probability) in the target model's output distribution. We test with a fixed $k = 3$ and $k = 5$.
- Entropy (Ours): Our Entropy-Aware Dynamic Verification.

As shown in Table III, when the verification threshold decreases progressively from *Greedy* to *Direct*, the draft model's acceleration effect on the target model improves, but at the cost of reduced generation quality. When the validation threshold is lowered, the target model accepts more edited content from the draft model, thereby improving generation speed. Because validation failure not only decreases the number of valid tokens but also forces the draft model to regenerate draft, impairing efficiency. However, due to the inherent differences in editing capabilities between the target model and the draft model, this efficiency gain comes at the expense of significantly degraded generation quality, evidenced by Pass@1 drops of approximately 6% and 13% on the CanItEdit. These findings reinforce our motivation: balancing acceleration and quality.

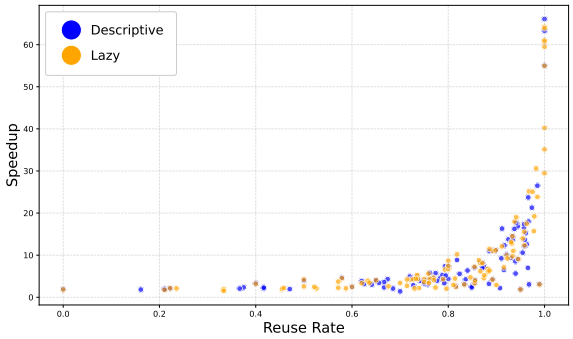
Our proposed validation method achieves this balance, delivering superior acceleration compared to the greedy algorithm while maintaining—and in some cases surpassing—its generation quality. We attribute this success to dynamic thresholding, which improves the draft acceptance rate, and effectively combines the editing strengths of both target and draft models to enhance quality. Notably, on the CodeIF-Bench, L-2 generation quality improves as the threshold decreases. We hypothesize that this results from the draft model's strong L-2 editing capabilities, further refined through supervised fine-tuning. Our validation method also optimally leverages this characteristic, maintaining generation quality between the draft and target models, thus further demonstrating its effectiveness.

> **RQ3 Summary:** Our *Entropy-Aware Dynamic Verification* method not only effectively improves the draft model's acceleration effect on the target model, but also maintains - and in some cases surpasses - the generation quality of greedy decoding validation.

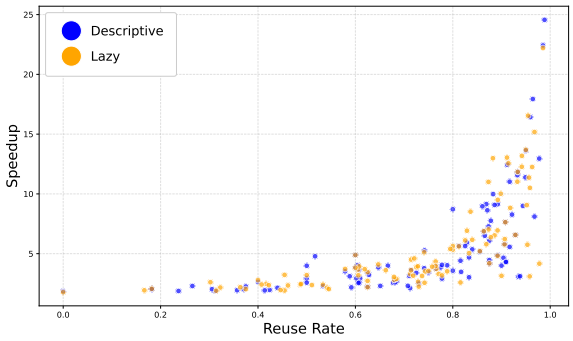
D. RQ4: Performance with Different Code Reuse Rates

We further investigate the impact of code reuse rates on EFFICIENTEDIT's performance. As shown in Figure 3, higher code reuse rates generally correlate with increased acceleration ratios. This is because reusable code is often block-based, and EFFICIENTEDIT's *Reuse* phase can efficiently reuse a large number of tokens in a single target model forward pass, thereby greatly improving efficiency. However, we observe that in a few cases, substantial increases in reuse rates lead to only marginal improvements in acceleration. We attribute this to the fragmentation of reusable code in these instances, which can result in only a small amount of code tokens being reused at a time and increase the overhead of switching between *Reuse* and *Generate*. Meanwhile, even at low reuse rates (e.g., $\tilde{0}.5$), our method achieves measurable acceleration, demonstrating robustness.

Performance varies significantly across different model families. The DeepSeek-Coder combination achieves peak acceleration exceeding 50× at a 100% reuse rate, while Qwen2.5-Coder's maximum acceleration remains below 25×. This disparity results in DeepSeek-Coder's average acceleration ratio appearing higher than Qwen2.5-Coder's at very high reuse rates. Our analysis suggests this anomaly (extremely high speedup at 100% reuse for DeepSeek-Coder) occurs partly because DeepSeek-Coder, being inherently somewhat weaker in



(a) DeepSeek-Coder



(b) Qwen2.5-Coder

Fig. 3: Inference speedup of EFFICIENTEDIT with different reuse rates on CanItEdit. The reuse rate is the percentage of tokens in the final edited code that come from the original code.

instruction following for certain complex edits than Qwen2.5-Coder in our experiments, occasionally tends to ignore editing instructions and output the unmodified code. In such 100% reuse scenarios, EFFICIENTEDIT very efficiently verifies the entire original code as correct with minimal overhead. Notably, when changing instructions from "lazy" to "descriptive" formats in CanItEdit, the frequency of 100% reuse cases for DeepSeek-Coder decreases significantly, indicating its greater sensitivity to instruction specificity. In contrast, Qwen2.5-Coder demonstrates superior robustness in adhering to edit instructions, rarely exhibiting 100% code reuse regardless of instruction style, leading to more consistent speedup patterns.

RQ4 Summary: In most scenarios, EFFICIENTEDIT demonstrates a positive correlation between code reuse rate and acceleration ratio, while still demonstrates good acceleration in lower reuse rate. Model-specific behaviors can influence performance at extreme reuse rates.

VI. DISCUSSION

A. Case Study

We demonstrate the effectiveness of our method through the case study presented in Figure 4. The task involves

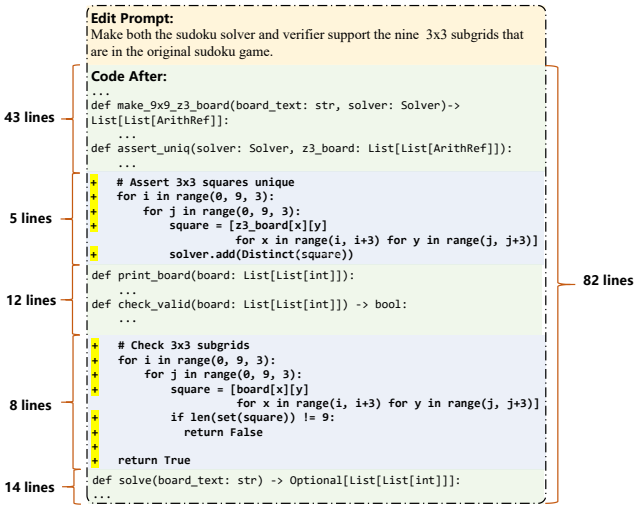


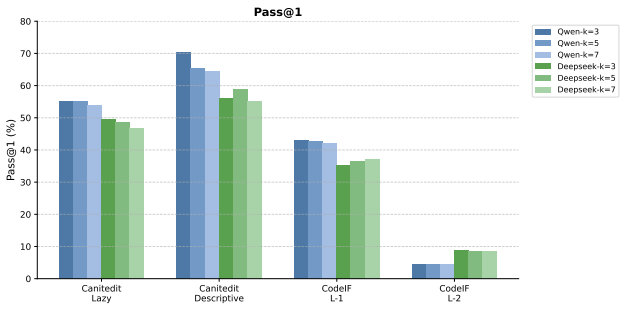
Fig. 4: Case study. An example generated by the Qwen2.5-Coder configuration from CanItEdit (task ID 25). The proportion of newly edited lines (patches) is approximately 16% of the final code.

implementing functional enhancements to existing code, representing a challenging code editing scenario where: (1) the precise editing locations cannot be determined solely from the instruction, and (2) multiple modification points are required. When applying EFFICIENTEDIT, the Qwen2.5-Coder configuration achieves a 5.2× speedup compared to traditional autoregressive methods, outperforming the best baseline (FastFixer at 3.3×) up to 50%. Qwen2.5-Coder-32B-Instructi requires 58.4 seconds for autoregressive decoding (using 4 RTX 4090 GPUs), while our method completes the same task in just 11.2 seconds, greatly improving editing efficiency. Our method provides superior performance through its two-phase design: (1) the *Reuse* phase reuses large amounts of redundant code in a lightweight manner while simultaneously identifying potential editing locations, and (2) the *Generate* phase enables efficient edit generation through high-quality draft models and an optimized verification algorithm that balances generation quality and efficiency.

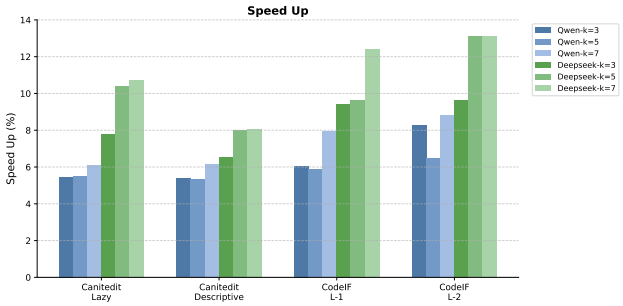
As illustrated in Figure 4, EFFICIENTEDIT achieves reuse of 69 lines of code through just 3 target model forward inferences in its *Reuse* phase, reusing 43 lines, 12 lines, and 14 lines, respectively. The first two reuse inferences also effectively pinpoint edit locations where new code needs to be inserted. Subsequently, the combination of a high-quality draft model (fine-tuned for edits) and our dynamic threshold verification mechanism enables efficient generation of the new code segments.

B. Hyperparameter Discussion

In EFFICIENTEDIT, the base threshold k plays a crucial role in the validation process. Figure 5 illustrates the performance of EFFICIENTEDIT with different threshold values ($k = 3, 5, 7$). The results reveal that: (1) the Qwen2.5-Coder configuration achieves optimal performance at $k = 3$, while



(a) Pass@1



(b) Speedup

Fig. 5: EFFICIENTEDIT performance (Pass@1 and Speedup) at different base entropy thresholds k.

the DeepSeek combination performs best at $k = 5$; (2) the acceleration ratio shows significant improvement with increasing threshold values; and (3) Pass@1 remains relatively stable across these tested thresholds, exhibiting only marginal variations. This indicates a degree of robustness in hyperparameter selection concerning generation quality within this range, allowing for tuning towards higher speedup without drastic quality drops. Our chosen values ($k = 3$ for Qwen, $k = 5$ for DeepSeek) represent a good balance found empirically.

C. Comparison with Code Editing Methods

Growing popularity of code editing techniques [35], [36], it is crucial to evaluate the efficiency of EFFICIENTEDIT in comparison with existing approaches. Specifically, we select the following two representative methods: the open-source code editing tool FastApply [36] and the pretraining-based approach CoEdPilot [35].

1) Compared to CoEdPilot: CoEdPilot is built on encoder-decoder architecture like CodeT5 [37], whereas EfficientEdit is designed for most LLMs based on decoder-only architecture (*e.g.*, Qwen, DeepSeek, LLaMA). This architectural mismatch makes direct effectiveness comparison challenging, and retraining CoEdPilot on larger decoder-only models is beyond our current scope. Thus, in Table IV, we compare editing time under the same experimental setting, acknowledging that CoEdPilot (about 1.xB parameters) and EfficientEdit (about 40B total) differ greatly in model-scale.

As seen from the results, EfficientEdit shows subtantially higher efficiency. Specifically, CoEdPilot uses a locator combined with previous location and editing operations to locate positions and uses a locator-generator loop with iterative forward passes, which introduces cumulative context and latency. Despite this, its localization-editing paradigm is promising. Since it still relies on autoregressive decoding, EfficientEdit or speculative decoding techniques could be integrated to further accelerate its pipeline.

TABLE IV: Editing time ($s \downarrow$) of CoEdPilot and EFFICIENTEDIT on CanItEdit dataset. The backbone is Qwen2.5-Coder combination.

	Lazy	**Descriptive**
CoEdPilot	40.2	38.2
EFFICIENTEDIT	**7.0**	**7.3**

TABLE V: Editing efficiency comparison of FastApply and EFFICIENTEDIT on the CanItEdit dataset. The backbone is Qwen2.5-Coder combination.

	Lazy		**Descriptive**	
	Token/s	Speedup	Token/s	Speedup
FastApply	18.9	1.45×	17.7	1.36×
EFFICIENTEDIT	**70.7**	**5.44**×	**70.0**	**5.38**×

2) Compared to FastApply: FastApply takes the original and diff-formatted code from a target-LLM as input and outputs full edited code. It achieves high throughput via commercial backends (*e.g.*, SoftGen [38]) that may optimize quantization, caching, or memory. To ensure fairness, we used the same environment as in our setting: Qwen2.5-Coder-32B-Instruct generated diff patches, and FastApply (7B), without any commercial tools, produced final outputs. Results are summarized as Table V. Under identical conditions, EfficientEdit achieves substantially higher token throughput and speedup than FastApply. While FastApply reduces redundancy via diff-based editing, it remains bottlenecked by autoregressive decoding, which generates one token per inference step. In contrast, EfficientEdit can verify and accept multiple tokens per step, breaking this limit and providing superior efficiency.

VII. RELATED WORK

A. LLM for Code Editing

Recent benchmarks (CanItEdit [6], EditEval [11], CodeEditorBench [13], SWE-Bench [39],CodeIF-Bench [31]) demonstrate substantial progress in LLM-based code editing [3], [4], [21]. Current approaches typically employ end-to-end training and inference paradigms, with models like StarCoder [21], OctoCoder [8], and EditCoder [6] leveraging commit histories in pre-training or fine-tuning to enhance editing capabilities. The recent development of InstructCoder [11], a model specifically designed and evaluated for instructional code editing tasks, represents a notable advancement in LLM-based code modification research. NextCoder [7] enhance its end-to-end code editing performance by incorporating a novel adaptive fine-tuning algorithm during training. However, their

efficiency remains constrained by autoregressive decoding limitations and generating redundant code. CodeEditor [12] enhances LLM-based code editing performance via additional pre-training on mutated code snippets. Some solutions like CoEdPilot [35] jointly train a localization LLM and a generation LLM, where the localization model pinpoints edit positions to improve the generation model's efficiency, but the autoregressive inference process in both models still constrains their overall editing efficiency. In contrast, we proposes the EFFICIENTEDIT method to improve the inference efficiency of autoregressive LLMs in code editing. Combining EFFICIENTEDIT maybe achieve more efficient and high-quality code editing.

B. Efficient Inference for LLMs

Most current LLMs rely on autoregressive decoding for inference, suffering from inefficient sequential token generation. Speculative decoding [22] has emerged as a promising solution, significantly improving inference speed while maintaining output quality. In general-purpose generation, approaches like lookahead decoding [25] employed phrases to accelerate the target model, while Ouroboros [24] uses rejected drafts and historical content to improve the efficiency of draft model generating drafts. Judge Decoding [40] enables the target model to recover correct but invalidated draft content via a trained discriminative module. For code-related tasks, REST [27] leverages the Stack dataset as a retrieval repository to boost generation efficiency, while FastFixer [28] retrieves reusable fragments from historical error codes as drafts to enhance program repair efficiency. Building upon these advances in speculative decoding and leveraging the unique characteristics of code editing tasks, we design a dedicated inference acceleration method specifically optimized for code editing scenarios.

VIII. THREATS TO VALIDITY

External Validity relates to the generalization and overhead of EFFICIENTEDIT. To assess generalization, we combine state-of-the-art backbones (Qwen and DeepSeek) and evaluate across function-, class-, and repository-level editing tasks on both CanItEdit and CodeIF-Bench. The results demonstrate EFFICIENTEDIT's excellent generalizability, thereby reducing threats to external validity. Regarding overhead, EFFICIENTEDIT incurs two additional costs: (1) extra GPU memory to load the draft model during inference, and (2) training cost for the draft model. Yet, the draft model uses only $\sim 22\%$ of the parameters and memory of the 32B target model, and training converges within 2 hours on two NVIDIA 4090 GPUs with training data easily generated by the target LLM. Despite this modest cost, EFFICIENTEDIT yields up to $8\times$ speedup. In low-resource settings, with any extra costs, the "Reuse" module alone delivers 3–$5\times$ acceleration.

Internal Validity relates to the hyperparameter-related threats and hardware environment. We conducted a focused analysis of the most influential base threshold k in our discussion section. This examination specifically elucidates base threshold k's impact on EFFICIENTEDIT's performance. Precise speedup numbers can be influenced by the specific hardware (GPUs, CPU) and software libraries used. While we report our setup, results might vary on different configurations.

Construct Validity relates to the metrics used in our evaluations. We use Pass@K for functional correctness and Tokens/s / Speedup for efficiency, which are standard in the field. However, Pass@K relies on test case execution and may not capture all aspects of code quality (e.g., readability, maintainability). Efficiency metrics, while objective, might not perfectly correlate with perceived developer productivity.

IX. CONCLUSION

In this work, we introduce EFFICIENTEDIT, an efficient acceleration method for LLM-based code editing. It employs a novel reuse-generate iterative paradigm via edit-oriented speculative decoding. EFFICIENTEDIT reuses code segments from code to be edited and locates potential editing positions to efficiently generate editing content through the edit-optimized draft model enhanced and the entropy-aware dynamic verification mechanism. Experimental results on a wide range of editing benchmarks show that EFFICIENTEDIT outperforms existing baselines in acceleration for code editing tasks, while maintaining—and in some cases improving—the quality of edited code.

ACKNOWLEDGEMENT

This research is supported by the National Natural Science Foundation of China Grants Nos. 62302021, 62272445, 62332001, CCF-Huawei Populus Grove Fund CCF-HuaweiSE202402, and the Fundamental Research Funds for the Central Universities (Grant No. JK2024-28).

REFERENCES

[1] J. A. OpenAI, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report, 2024," *URL https://arxiv. org/abs/2303.08774*, vol. 2, p. 6, 2024.

[2] claude, "Claude," 2023. [Online]. Available: https://claude.ai/

[3] Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma *et al.*, "Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence," *arXiv preprint arXiv:2406.11931*, 2024.

[4] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, K. Dang, Y. Fan, Y. Zhang, A. Yang, R. Men, F. Huang, B. Zheng, Y. Miao, S. Quan, Y. Feng, X. Ren, X. Ren, J. Zhou, and J. Lin, "Qwen2.5-coder technical report," 2024. [Online]. Available: https://arxiv.org/abs/2409.12186

[5] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[6] F. Cassano, L. Li, A. Sethi, N. Shinn, A. Brennan-Jones, J. Ginesin, E. Berman, G. Chakhnashvili, A. Lozhkov, C. J. Anderson, and A. Guha, "Can it edit? evaluating the ability of large language models to follow code editing instructions," in *First Conference on Language Modeling*, 2024. [Online]. Available: https://openreview.net/forum?id=D06yk3DBas

[7] T. Aggarwal, S. Singh, A. Awasthi, A. Kanade, and N. Natarajan, "Nextcoder: Robust adaptation of code LMs to diverse code edits," in *ICLR 2025 Third Workshop on Deep Learning for Code*, 2025. [Online]. Available: https://openreview.net/forum?id=r1kOC0MiyG

[8] N. Muennighoff, Q. Liu, A. R. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. V. Werra, and S. Longpre, "Octopack: Instruction tuning code large language models," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=mw1PWNSWZP

[9] Github, "Github copilot," 2021. [Online]. Available: https://github.com/features/copilot

[10] Anysphere, "Cursor," 2023. [Online]. Available: https://www.cursor.com/

[11] K. Li, Q. Hu, J. Zhao, H. Chen, Y. Xie, T. Liu, M. Shieh, and J. He, "Instructcoder: Instruction tuning large language models for code editing," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, 2024, pp. 50–70.

[12] J. Li, G. Li, Z. Li, Z. Jin, X. Hu, K. Zhang, and Z. Fu, "Codeeditor: Learning to edit source code with pre-trained models," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–22, 2023.

[13] J. Guo, Z. Li, X. Liu, K. Ma, T. Zheng, Z. Yu, D. Pan, Y. Li, R. Liu, Y. Wang *et al.*, "Codeeditorbench: Evaluating code editing capability of large language models," *arXiv preprint arXiv:2404.03543*, 2024.

[14] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

[15] R. Kitchin and M. Dodge, *Code/space: Software and everyday life*. Mit Press, 2014.

[16] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, "Reading between the lines: Modeling user behavior and costs in ai-assisted programming," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–16.

[17] X. Chen, X. Hu, Y. Huang, H. Jiang, W. Ji, Y. Jiang, Y. Jiang, B. Liu, H. Liu, X. Li, X. Lian, G. Meng, X. Peng, H. Sun, L. Shi, B. Wang, C. Wang, J. Wang, T. Wang, J. Xuan, X. Xia, Y. Yang, Y. Yang, L. Zhang, Y. Zhou, and L. Zhang, "Deep learning-based software engineering: Progress, challenges, and opportunities," *Science China Information Sciences*, vol. 68, no. 1, p. 111102, 2025. [Online]. Available: http://www.sciengine.com/publisher/Science-China-Press/journal/SCIENCE-CHINA-Information-Sciences/68/1/10.1007/s11432-023-4127-5

[18] B. Lin, S. Wang, Z. Liu, Y. Liu, X. Xia, and X. Mao, "Cct5: A code-change-oriented pre-trained model," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1509–1521.

[19] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "Coditt5: Pretraining for source code and natural language editing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[20] J. Wei, G. Durrett, and I. Dillig, "Coeditor: Leveraging repo-level diffs for code auto-editing," in *ICLR*. OpenReview.net, 2024.

[21] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[22] Y. Leviathan, M. Kalman, and Y. Matias, "Fast inference from transformers via speculative decoding," 2023. [Online]. Available: https://arxiv.org/abs/2211.17192

[23] C. Chen, S. Borgeaud, G. Irving, J.-B. Lespiau, L. Sifre, and J. Jumper, "Accelerating large language model decoding with speculative sampling," 2023. [Online]. Available: https://arxiv.org/abs/2302.01318

[24] W. Zhao, Y. Huang, X. Han, W. Xu, C. Xiao, X. Zhang, Y. Fang, K. Zhang, Z. Liu, and M. Sun, "Ouroboros: Generating longer drafts phrase by phrase for faster speculative decoding," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024, pp. 13 378–13 393.

[25] Y. Fu, P. Bailis, I. Stoica, and H. Zhang, "Break the sequential dependency of llm inference using lookahead decoding," in *Proceedings of the 41st International Conference on Machine Learning*, 2024, pp. 14 060–14 079.

[26] J. Zhang, J. Wang, H. Li, L. Shou, K. Chen, G. Chen, and S. Mehrotra, "Draft& verify: Lossless large language model acceleration via self-speculative decoding," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 11 263–11 282.

[27] Z. He, Z. Zhong, T. Cai, J. Lee, and D. He, "REST: Retrieval-based speculative decoding," in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, K. Duh, H. Gomez, and S. Bethard, Eds. Mexico City, Mexico: Association for Computational Linguistics, Jun. 2024, pp. 1582–1595. [Online]. Available: https://aclanthology.org/2024.naacl-long.88/

[28] F. Liu, Z. Liu, Q. Zhao, J. Jiang, L. Zhang, Z. Sun, G. Li, Z. Li, and Y. Ma, "Fastfixer: An efficient and effective approach for repairing programming assignments," in *ASE*, 2024, pp. 669–680. [Online]. Available: https://doi.org/10.1145/3691620.3695062

[29] J. Xu, X. Liu, J. Yan, D. Cai, H. Li, and J. Li, "Learning to break the loop: Analyzing and mitigating repetitions for neural text generation," *Advances in Neural Information Processing Systems*, vol. 35, pp. 3082–3095, 2022.

[30] Y. Dong, Y. Liu, X. Jiang, Z. Jin, and G. Li, "Rethinking repetition problems of llms in code generation," *arXiv preprint arXiv:2505.10402*, 2025.

[31] P. Wang, L. Zhang, F. Liu, L. Shi, M. Li, B. Shen, and A. Fu, "Codeif-bench: Evaluating instruction-following capabilities of large language models in interactive code generation," 2025. [Online]. Available: https://arxiv.org/abs/2503.22688

[32] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi, "Self-instruct: Aligning language models with self-generated instructions," in *ACL*, 2023.

[33] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[34] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[35] C. Liu, Y. Cai, Y. Lin, Y. Huang, Y. Pei, B. Jiang, P. Yang, J. S. Dong, and H. Mei, "Coedpilot: Recommending code edits with learned prior edit relevance, project-wise awareness, and interactive nature," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 466–478.

[36] FastApply, "Fastapply," 2024. [Online]. Available: https://huggingface.co/Kortix/FastApply-1.5B-v1.0

[37] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[38] SoftGen, "Softgen." [Online]. Available: https://softgen.ai/

[39] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" in *ICLR*, 2024.

[40] G. Bachmann, S. Anagnostidis, A. Pumarola, M. Georgopoulos, A. Sanakoyeu, Y. Du, E. Schönfeld, A. Thabet, and J. Kohler, "Judge decoding: Faster speculative sampling requires going beyond model alignment," 2025. [Online]. Available: https://arxiv.org/abs/2501.19309