LET THE CODE LLM EDIT ITSELF WHEN YOU EDIT THE CODE

Anonymous authors
Paper under double-blind review

ABSTRACT

In this work, we investigate a typical scenario in code generation where a developer edits existing code in real time and requests a code assistant, e.g., a large language model, to re-predict the next token or next line on the fly. Naively, the LLM needs to re-encode the entire KV cache to provide an accurate prediction. However, this process is computationally expensive, especially when the sequence length is long. Simply encoding the edited subsequence and integrating it to the original KV cache meets the temporal confusion problem, leading to significantly worse performance. We address this efficiency and accuracy trade-off by introducing **P**ositional **I**ntegrity **E**ncoding (PIE). Building upon the rotary positional encoding, PIE first removes the rotary matrices in the Key cache that introduce temporal confusion and then reapplies the correct rotary matrices. This process ensures that positional relationships between tokens are correct and requires only a single round of matrix multiplication. We validate the effectiveness of PIE through extensive experiments on the RepoBench-C-8k dataset, utilizing DeepSeek-Coder models with 1.3B, 6.7B, and 33B parameters. Our evaluation includes three real-world coding tasks: code insertion, code deletion, and multi-place code editing. Results demonstrate that PIE reduces computational overhead by over 85% compared to the standard full recomputation approach across all model sizes and tasks while well approximating the model performance.

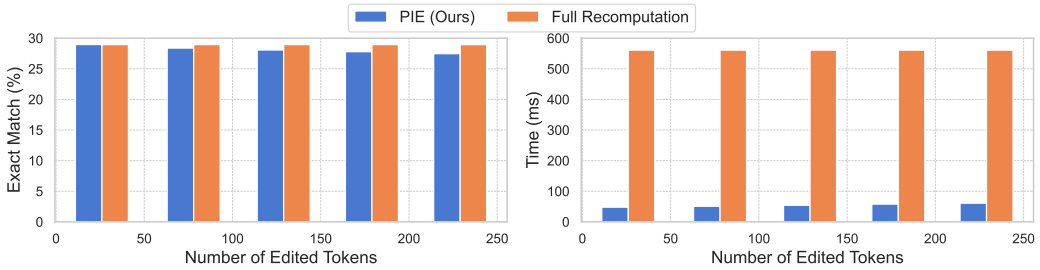

Figure 1: Latency and accuracy comparison of the full recomputation approach and our PIE using DeepSeek-Coder 6.7B on the RepoBench-C-8k(XF-F) Python dataset on a single A100 GPU. The latency only records the time cost for the KV cache update.

1 INTRODUCTION

Large language models (LLMs) (Dettmers et al., 2022; Anil et al., 2023; Touvron et al., 2023; Zeng et al., 2023) have seen widespread adoption and achieved impressive results across various natural language processing (NLP) tasks. Despite these successes, LLMs face significant computational challenges, particularly in handling long sequences. To address this, numerous approaches have been proposed to accelerate the inference process, including lossless (e.g., memory and IO optimization (Dao et al., 2022; Kwon et al., 2023; Sheng et al., 2023), speculative decoding (Stern et al., 2018; Leviathan et al., 2023)) and lossy techniques (e.g., quantization (Frantar et al., 2022; Xiao et al., 2023) and KV cache eviction (Xiao et al., 2024; Zhang et al., 2024)). We refer to the above setting as

the *static* setting, where the content is fixed, and the goal is to generate responses efficiently without compromising too much on performance.

Besides the static setting, we observe there is a strong demand for an alternative, which we call the *real-time editing* setting, where users frequently edit the content and expect the LLM to generate correct responses based on the updated information. A typical scenario is the interactive coding assistant, where developers often make incremental changes to their existing code and require the AI copilot to correctly predict the next line or complete a partial code snippet on the fly. The standard approach is re-encoding the KV cache of the content after each edit and then making the prediction. However, as illustrated in Figure 1, this approach leads to considerable substantial computational overhead and latency when the content is long (Fu, 2024; Agrawal et al., 2024), making it impractical for real-time applications where quick and accurate responses are essential.

In this paper, we aim to improve the efficiency of AI copilots in real-time editing scenarios, as illustrated in Figure 2. Naively, an efficient strategy is encoding only the edited subsequence and then directly integrating those keys and values into the original KV cache. However, this strategy results in temporal confusion between the pre-edit and post-edit sequences. Keys of certain positions either disappear or multiple keys at different positions share the same index, causing the model to attend to incorrect information, leading to poor next-token prediction performance in practice. To address this problem, we introduce **P**ositional **I**ntegrity **E**ncoding (PIE). PIE is built upon rotary positional encoding (RoPE) (Su et al., 2021), the de-facto standard component in modern LLMs. PIE first removes the rotary matrices in the Key cache that introduce temporal confusion and then reapplies the correct rotary matrix for each position through simple matrix multiplications. By ensuring that the positional relationships between tokens in the new sequence are unique and consecutive, PIE can help the model make accurate predictions. It is worth noting that the calculation of PIE requires only a single round of matrix operations to modify the KV cache, resulting in negligible computational overhead.

We demonstrate the effectiveness of Positional Integrity Encoding (PIE) through extensive experiments conducted on the RepoBench-C-8k dataset, utilizing the DeepSeek-Coder (Guo et al., 2024) models with 1.3B, 6.7B, and 33B parameters. To rigorously evaluate PIE's performance, we curated three tasks designed to simulate real-world coding scenarios: code insertion, code deletion, and multi-place code edition. These tasks were chosen to reflect common operations that developers perform during interactive coding sessions, thereby providing a comprehensive assessment of PIE's practical utility. Our experimental results indicate that PIE achieves a reduction in computational overhead of over 85% for editing the KV cache across all model sizes and tasks without compromising performance compared to the naive full-recomputation approach. By leveraging PIE, developers can experience efficient interactions with AI coding assistants.

2 Related Work

Positional Encodings Positional information is essential for modeling languages. The original Transformer model (Vaswani et al., 2017) encodes positional information using Absolute Positional Encoding (APE). In particular, a (learnable) real-valued embedding is assigned to each position i. Differently, Relative Positional Encodings (RPE) (Shaw et al., 2018; Dai et al., 2019; Raffel et al., 2020; Press et al., 2022; Su et al., 2021; Luo et al., 2021; 2022; Chi et al., 2022; Sun et al., 2023; Chi et al., 2023; Li et al., 2023; He et al., 2024) instead encode the relative distance $i - j$ for each position pair (i, j). One of the most widely used RPE in state-of-the-art LLMs is Rotary Position Encoding (RoPE) (Su et al., 2021). RoPE rotates the query and key vectors by an angle proportional to their absolute positions before the attention mechanism, resulting in the attention being a function of the relative distance between tokens.

In the literature, relative positional encodings play essential roles across various tasks and data modalities, such as improving the length extrapolation capability of language models (Press et al., 2022; Sun et al., 2023; Chi et al., 2022; 2023; He et al., 2024) and enabling flexible modeling of structural information beyond sequence data like images (Liu et al., 2021) and graphs (Ying et al., 2021; Zhang et al., 2023a; Luo et al., 2023). In this work, we develop the Positional Integrity Encoding based on RoPE to improve the efficiency of LLMs in the real-time editing setting.

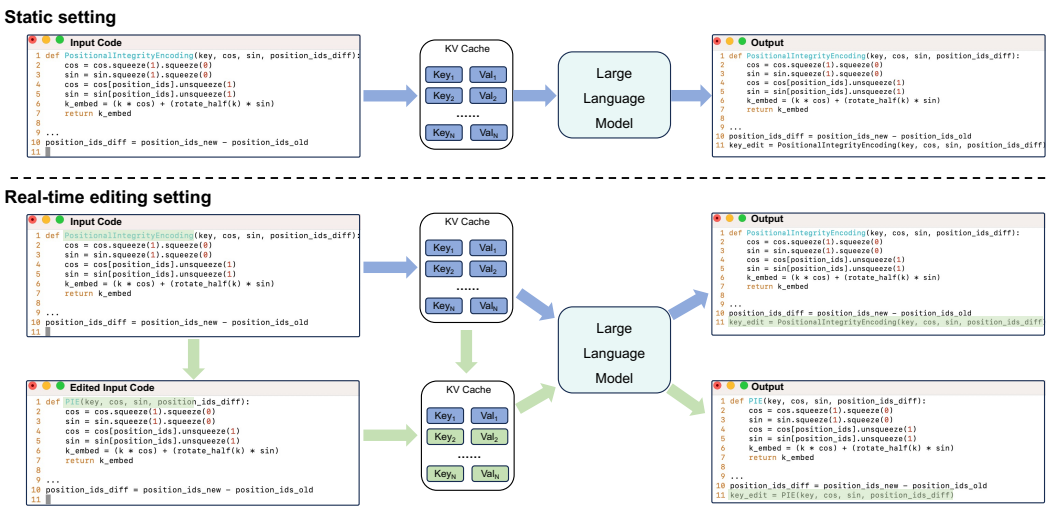

Figure 2: Illustration of the KV cache mechanism in both static and real-time editing settings for large language models (LLMs). **Top:** In the static setting, the model processes a fixed input to generate predictions, leveraging precomputed Key/Value (KV) pairs stored in the cache. **Bottom:** In the real-time editing setting, the input code is frequently edited, necessitating updates to the KV cache to maintain accurate information to generate the correct next tokens. Our objective is to optimize the efficiency of the green arrow pathway, which represents the process of updating the KV cache in response to code edits.

Transformer Efficiency Improving the efficiency of Transformer models has great significance in real-world applications. In the literature, existing approaches can be briefly categorized into (1) efficient attention, (2) model compression, and (3) system-architecture co-design.

The attention module of Transformer needs to calculate pairwise correlations between all positions, resulting in quadratic time and memory cost with respect to the sequence length. To reduce the cost, many efficient attention variants have been proposed, such as (1) sparse attentions (Child et al., 2019; Beltagy et al., 2020; Qiu et al., 2020), which design either pre-defined or learnable patterns to reduce the amount of the key-value pairs that each query needs to attend to; (2) approximation-based attention (Katharopoulos et al., 2020; Wang et al., 2020; Choromanski et al., 2021; Kitaev et al., 2020; Tay et al., 2020; Roy et al., 2021), which use tailored approaches like low-rank projection or random features to approximate standard attention for efficient computation.

Another perspective for Transformer efficiency is model compression, mainly including (1) pruning (Wang et al., 2021; Hubara et al., 2021; Ma et al., 2023; Frantar & Alistarh, 2023), which aims at removing redundant model parameters or layers for efficient deployment without scarifying performance; (2) quantization (Yao et al., 2022; Park et al., 2022; Dettmers et al., 2022; Frantar et al., 2022; Xiao et al., 2023; Liu et al., 2023), which uses post-processing to represent weights and activations via low-precision format for reducing time and memory costs; (3) knowledge distillation (Sanh et al., 2019; Gu et al., 2024), which uses a smaller model to learn knowledge from a large model for balancing efficiency-accuracy trade-offs.

In the era of LLMs, the importance of system-architecture co-design has been highlighted to improve Transformer efficiency further. Many works begin to design more efficient approaches for serving Transformers regarding the characteristics of computer systems for real-world applications, such as FlashAttention (Dao et al., 2022; Dao, 2023), PagedAttention (Kwon et al., 2023), and FlexGen (Sheng et al., 2023) that are proposed for memory and I/O optimization. Additionally, to reduce functional calls during generation, speculative decoding (Stern et al., 2018; Leviathan et al., 2023; Chen et al., 2023; Miao et al., 2023; Spector & Re, 2023; Cai et al., 2024; Zhang et al., 2023b; He et al., 2023; Li et al., 2024) has been proposed. Our Positional Integrity Encoding is specially designed to improve the efficiency of LLMs in real-time editing settings, which can be seamlessly combined with all the above-introduced approaches to achieve further speed-ups.

3 METHODS

3.1 BACKGROUND

Let $s = (w_1, w_2, \ldots, w_n)$ represent the input token sequence, where each w_i belongs to a fixed vocabulary. Let θ_{LLM} represent a Transformer-based large language model, which can calculate the conditional probability distribution of the next token $p(w_{n+1}|s; \theta_{\text{LLM}})$ and generate tokens iteratively. Typically, the input s is fixed. Therefore, the generation process of LLMs usually employs the KV Cache mechanism (Pope et al., 2023) to store previously computed Key/Value vectors during each layer's attention calculation. We denote the KV cache as $\boldsymbol{K} = (K_1, K_2, \ldots, K_n)$ and $\boldsymbol{V} = (V_1, V_2, \ldots, V_n)$, where K_i and V_i are the keys and values associated with token w_i. When predicting the token at position $n+1$, we can use w_n as input and, in each layer, compute the attention between the current hidden representation and the stored KV cache, avoiding recomputing the hidden representation of previous tokens. Without any confusion, we also denote the next token probability distribution as $p(w_{n+1}|\boldsymbol{K}, \boldsymbol{V}, w_n; \theta_{\text{LLM}})$.

In this study, we aim to investigate a new scenario where the context s is real-time edited by users, which makes it impossible for the KV cache to predict the correct next token without any modification. We can model such real-time context as a sequence of steps. Each step can be formulated as an action where tokens from position i to j in s are edited, resulting in a modified sequence $s^{\text{edit}} = [w_1, \ldots, w_i, a_1, a_2, \ldots, a_m, w_{j+1}, \ldots, w_n]$, where $[a_1, a_2, \ldots, a_m]$ represent the new inputs that replace $[w_i, \ldots, w_j]$. Our goal is to accurately and efficiently predict w_{n+1} given by θ_{LLM} on s^{edit}. This problem is crucial in various scenarios. For instance, users can frequently edit their previous codes for different purposes and expect the code language model to swiftly adapt to these changes and predict the correct next line based on the updated information.

As the context between position i and position j is edited, the KV cache corresponding to these tokens must be updated. Furthermore, these changes will impact the representations of subsequent tokens after position j, thereby necessitating updates to all subsequent KV cache. The naive approach involves a full-recomputation strategy: re-encoding all the KV cache for $[a_1, a_2, \ldots, a_m, w_{j+1}, \ldots, w_n]$ layer by layer, followed by making predictions using the updated cache $\boldsymbol{K}^*$ and $\boldsymbol{V}^*$. This approach ensures the KV cache is exact when predicting the next tokens. However, it is easy to see that it is computationally expensive, especially when the edits are light but the texts to be re-encoded are long. It's worth noting that the original $\boldsymbol{K}$ and $\boldsymbol{V}$ already encode rich information on $[w_{j+1}, \ldots, w_n]$, and a full recomputation may not be essential for practical problems. With this in mind, we seek to find ways to efficiently edit $\boldsymbol{K}$ and $\boldsymbol{V}$, yielding $\boldsymbol{K}^{\text{edit}}$ and $\boldsymbol{V}^{\text{edit}}$, which approximates $p(w_{n+1}|\boldsymbol{K}^*, \boldsymbol{V}^*, w_n; \theta_{\text{LLM}}) \approx p(w_{n+1}|\boldsymbol{K}^{\text{edit}}, \boldsymbol{V}^{\text{edit}}, w_n; \theta_{\text{LLM}})$ in an effective way.

3.2 POSITIONAL INTEGRITY ENCODING (PIE)

When a user modifies s into s^{edit}, KV cache associated with the first i tokens, i.e., $\boldsymbol{K}_{[1:i]}$ and $\boldsymbol{V}_{[1:i]}$, remains unchanged. As $[a_1, a_2, \ldots, a_m]$ is the user's new input, we feed this subsequence to the LLM to obtain the keys and values from position $i+1$ to $i+m$. We denote this piece of new KV cache as $\boldsymbol{K}^{\text{edit}}_{[i+1:i+m]}$ and $\boldsymbol{V}^{\text{edit}}_{[i+1:i+m]}$, and now have the edited KV cache as:

$$\boldsymbol{K}^{\text{edit}} = \text{Concat}(\boldsymbol{K}_{[1:i]}, \boldsymbol{K}^{\text{edit}}_{[i+1:i+m]}, \boldsymbol{K}_{[j+1:n]}), \tag{1}$$

$$\boldsymbol{V}^{\text{edit}} = \text{Concat}(\boldsymbol{V}_{[1:i]}, \boldsymbol{V}^{\text{edit}}_{[i+1:i+m]}, \boldsymbol{V}_{[j+1:n]}), \tag{2}$$

where the red symbols indicate real-time calculations.

Challenges. The key challenge lies in how to edit the succeeding KV cache $\boldsymbol{K}_{[j+1:n]}$ and $\boldsymbol{V}_{[j+1:n]}$. Clearly, the modification of $[a_1, a_2, \ldots, a_m]$ impacts the subsequent content in two ways: semantically and structurally. The semantic impact refers to the changes in the understanding of the subsequent text caused by the edited content. This can be a problem in natural language applications, such as dialog systems, where modifications to earlier conversations can significantly influence the generation of current responses. The other impact is structural, primarily concerning the temporal confusion between the pre-edit and post-edit sequences when $j - i \neq m$. This issue arises with common editing actions in code, such as additions and deletions (corresponding to $j - i = 0$ or $m = 0$). To be more concrete, imagine the original sequence has 5 tokens. If we add three tokens

between the second and third position, it will occupy the positional index $[3, 4, 5]$. We calculate $\boldsymbol{K}^{\text{edit}}_{[3:5]}$ and $\boldsymbol{V}^{\text{edit}}_{[3:5]}$ in equation (1) and (2). However, in the original $\boldsymbol{K}_{[3:5]}$ and $\boldsymbol{V}_{[3:5]}$, the positional index $[3, 4, 5]$ is also occupied. If we integrate them together and take no actions during the next token prediction, the model will calculate similarity with multiple keys in the KV cache with the same index $[3, 4, 5]$, causing confusion and potential prediction errors. Empirically, in code tasks, we find that the semantic impact is relatively small. Addressing temporal confusion for light edits alone can already lead to good performance (see the experiments in Section 4 for more details).

Our approach. To mitigate the temporal confusion during real-time editing, we propose a simple yet effective solution: **P**ositional **I**ntegrity **E**ncoding (PIE), which ensures that positional information remains correctly ordered after editing without the need to re-encode the KV cache for subsequent tokens. PIE builds upon the rotary positional encoding (RoPE) (Su et al., 2021), which is the most widely used positional encoding in LLMs. Without loss of generality, given a query vector $\boldsymbol{x}_i$ at position i and a key vector $\boldsymbol{x}_j$ at position j, RoPE calculates the dot-product similarity using

$$z_{ij} = \boldsymbol{x}_i^T W_q^T \boldsymbol{R}_{j-i} W_k \boldsymbol{x}_j \tag{3}$$

where $\boldsymbol{R}_{j-i}$ is the rotary matrix parameterized by the relative distance $j - i$, and W_q and W_k are learnable projection matrices. By definition, $\boldsymbol{R}_{j-i}$ can be expressed by the multiplication of two rotary matrices:

$$\boldsymbol{R}_{j-i} = \boldsymbol{R}_i^T \boldsymbol{R}_j \tag{4}$$

For practical implementation, during inference, we compute $\boldsymbol{R}_{\Theta,i} W_k x_i$ as the key on the fly and store it in the cache, and when a query arrives at a new position, we rotate the query using its corresponding rotary matrix and calculate its similarity with all the keys in the cache to obtain the attention scores.

It can be easily seen that the positional information in the KV cache is encoded within the rotary matrix. When an edit occurs, the rotary matrix associated with the keys must be adjusted to reflect their post-edit locations. Leveraging the formulation of RoPE-based attention calculation, this challenge can be addressed by first removing the rotary matrices in $\boldsymbol{K}$ that introduce temporal confusion and then reapplying the correct rotary matrix. In detail, assume we would like to update the key vector $\boldsymbol{k}^l_{j'}$ for the original position $j' \in [j+1, n]$, where $l \in [1, L]$ is the layer index. We can simply edit the key vector by using

$$\boldsymbol{k}^{\text{edit},l}_{j'} = \boldsymbol{R}_{i+m+j'-j} \boldsymbol{R}^{-1}_{j'} \boldsymbol{k}^l_{j'} \tag{5}$$

where $\boldsymbol{R}^{-1}_{j'}$, the inverse rotary matrix at position j', is used to remove the incorrect positional information, and $\boldsymbol{R}_{i+m+j'-j}$ is used to encode the correct position $i + m + j' - j$ in s^{edit}. It can easily seen that the computation can be further simplified as

$$\boldsymbol{k}^{\text{edit},l}_{j'} = \boldsymbol{R}_{i+m+j'-j} \boldsymbol{R}^{-1}_{j'} \boldsymbol{k}^l_{j'} = \boldsymbol{R}_{i+m+j'-j} \boldsymbol{R}_{-j'} \boldsymbol{k}^l_{j'} = \boldsymbol{R}_{i+m-j} \boldsymbol{k}^l_{j'} \tag{6}$$

Hence, the full editing process for $\boldsymbol{K}_{[j+1:n]}$ is as follows:

$$\begin{aligned} \boldsymbol{K}^{\text{edit}}_{[j+1:n]} &= [K^{\text{edit}}_{j+1}, \ldots, K^{\text{edit}}_n] \\ \text{where each } K^{\text{edit}}_{j'} &= \{\boldsymbol{k}^{\text{edit},1}_{j'}, \cdots, \boldsymbol{k}^{\text{edit},l}_{j'}, \cdots, \boldsymbol{k}^{\text{edit},L}_{j'}\}, j' \in [j+1, n], l \in [1, L] \\ \text{each } \boldsymbol{k}^{\text{edit},l}_{j'} &= \boldsymbol{R}_{i+m-j} \boldsymbol{k}^l_{j'} \end{aligned} \tag{7}$$

Unlike the full recomputation approach, the above calculation only requires a single round of matrix multiplication to directly modify the pre-computed KV cache, where the computational overhead can be considered negligible. By utilizing these transformations, we finally construct the edited KV cache as:

$$\boldsymbol{K}^{\text{edit}} = \text{Concat}(\boldsymbol{K}_{[1:i]}, \boldsymbol{K}^{\text{edit}}_{[i+1:i+m]}, \boldsymbol{K}^{\text{edit}}_{[j+1:n]}), \tag{8}$$

$$\boldsymbol{V}^{\text{edit}} = \text{Concat}(\boldsymbol{V}_{[1:i]}, \boldsymbol{V}^{\text{edit}}_{[i+1:i+m]}, \boldsymbol{V}_{[j+1:n]}), \tag{9}$$

where the red symbols indicate real-time calculations. The LLM then makes predictions based on $p(x_{n+1}|\boldsymbol{K}^{\text{edit}}, \boldsymbol{V}^{\text{edit}}, x_n; \theta_{\text{LLM}})$. It is worth noting that PIE is compatible with KV cache eviction methods (Xiao et al., 2024; Zhang et al., 2024; Liu et al., 2024b). These KV cache eviction methods focus on reducing the memory usage of the KV cache during inference. PIE is designed to obtain the KV cache of the edited context with minimal overhead. By integrating PIE with KV cache eviction methods, it is possible to maintain efficient memory management while ensuring the integrity of the positional information in the real-time edit setting.

Table 1: Statistics of RepoBench-C-8k (Liu et al., 2024a) test set.

Language	XF-F	XF-R	IF	Average Number of Tokens
Python	18,000	7,500	10,500	3,967
Java	18,000	7,500	10,500	4,179

4 EXPERIMENTS

In this section, we empirically study the effectiveness of our proposed method. In particular, we aim at answering the following questions through experiments:

- **Question 1**: Can our Positional Integrity Encoding maintain the prediction accuracy of full re-computation in code editing scenarios?
- **Question 2**: How much efficiency improvement can be achieved by using our Positional Integrity Encoding compared to existing approaches?
- **Question 3**: How large is the gap between our Positional Integrity Encoding and full re-computation in terms of LLM's predictions & representations?

We will answer each question with carefully designed experiments in the following sub-sections.

4.1 EXPERIMENTAL SETUP

Tasks. Our experiments are conducted on RepoBench-C-8k (Liu et al., 2024a). This benchmark focuses on the prediction of the next line of code, given a set of in-file context (including import statements and preceding lines before the target line), and cross-file context (comprising snippets from other files parsed by import statements). The detailed statistics of RepoBench-C-8k is shown in Table 1. To effectively evaluate next-line prediction performance of code LLMs, we follow Liu et al. (2024a) to use three task settings: (1) Cross-File-First (XF-F): mask the first appearance of a cross-file line within a file; (2) Cross-File-Random (XF-R): mask a random and non-first occurrence of a cross-file line; (3) In-File (IF): mask an in-file line that does not involve any cross-file modules. Moreover, we carefully design three real-world scenarios covering code insertion, code deletion, and code edition to comprehensively examine our approach. See Appendix A.1 for more detailed descriptions of tasks construction.

Settings. In our experiments, we employ DeepSeek-Coder (Guo et al., 2024), a code LLM that achieves strong performance in handling repository-level code completion tasks (We also conduct experiments on CodeLlama (Roziere et al., 2023) in Appendix A.2). We use Transformers (Wolf et al., 2020) as our codebase. We benchmark our method on models of different sizes covering 1.3B, 6.7B, and 33B. During inference, the greedy decoding strategy is used to deterministically generate 64 tokens. For 1.3B and 6.7B models, all the experiments are conducted on a single NVIDIA A100 GPU. For 33B models, the time for encoding the context is conducted on two NVIDIA A100 GPUs and the full generation process is conducted on eight NVIDIA A100 GPUs. The first non-comment line in the output is truncated and used as the prediction. The batch size is set to 1. All experiments are repeated three times with different seeds and the averaged scores are reported.

Evaluation. For comparison with our Positional Integrity Encoding, we choose two standard approaches as baselines: (1) Full-recomputation: re-compute the KV cache for all edited tokens and subsequent tokens; (2) Conflict Fast Encoding: re-compute the KV cache for the edited tokens while keeping the rest of the cache intact (i.e., using equation (1,2). Following Lu et al. (2021), we use Exact Match (EM) and Edit Similarity (ES) (Svyatkovskiy et al., 2020) to evaluate the accuracy of the predicted code lines on code completion tasks. We also report the time required to encode the edited context for efficiency evaluation.

4.2 MAIN RESULTS

Positional Integrity Encoding perfectly preserves the full re-computation performance. Results of different code editing settings are presented in Table 2, 3 and 4 respectively. It can be easily seen

Table 2: **Performance comparisons of insertion experiments.** In this task, for each next-line prediction target, we insert several lines of code into its context randomly to simulate real-world scenarios. EM and ES denote the Exact Match and Edit Similarity score respectively. All results demonstrate that our Positional Integrity Encoding approach brings substantial speed-ups without performance drops.

	Model	Method	XF-F			XF-R			IF		
			EM	ES	Time	EM	ES	Time	EM	ES	Time
Python	1.3B	Full-recomputation	22.42	65.26	192ms	35.41	72.96	193ms	28.78	69.22	193ms
Python	1.3B	Conflict Fast Encoding	7.32	43.73	23ms	10.61	47.18	22ms	8.91	45.25	22ms
Python	1.3B	PIE	22.3	65.2	29ms	35.33	72.88	28ms	28.69	69.13	29ms
Python	6.7B	Full-recomputation	28.95	70.11	561ms	40.89	76.19	564ms	35.26	72.73	562ms
Python	6.7B	Conflict Fast Encoding	5.35	33.32	34ms	6.52	35.25	34ms	6.09	38.76	34ms
Python	6.7B	PIE	28.83	70.01	50ms	40.77	76.14	50ms	35.2	72.72	50ms
Python	33B	Full-recomputation	35.75	73.46	2194ms	46.0	78.9	2199ms	39.75	75.12	2194ms
Python	33B	Conflict Fast Encoding	3.96	30.13	126ms	5.41	32.32	121ms	3.92	35.56	127ms
Python	33B	PIE	35.77	73.45	134ms	45.74	78.85	140ms	39.74	75.1	141ms
Java	1.3B	Full-recomputation	26.21	70.89	200ms	36.77	76.31	200ms	45.89	78.04	198ms
Java	1.3B	Conflict Fast Encoding	0.29	3.12	22ms	0.57	3.19	23ms	0.7	2.63	23ms
Java	1.3B	PIE	26.13	70.82	30ms	36.67	76.25	30ms	45.92	77.99	29ms
Java	6.7B	Full-recomputation	32.51	75.56	578ms	41.97	79.41	578ms	50.86	80.53	578ms
Java	6.7B	Conflict Fast Encoding	0.47	2.77	34ms	0.76	2.78	35ms	0.67	2.48	33ms
Java	6.7B	PIE	32.21	75.47	50ms	41.96	79.32	49ms	50.85	80.43	48ms
Java	33B	Full-recomputation	35.05	76.93	2269ms	44.95	80.87	2281ms	53.23	81.76	2270ms
Java	33B	Conflict Fast Encoding	0.38	2.51	120ms	0.59	2.40	122ms	0.68	2.09	122ms
Java	33B	PIE	34.78	76.78	138ms	45.01	80.95	133ms	53.16	81.66	139ms

Table 3: **Performance comparisons of deletion experiments.** In this task, for each next-line prediction target, we delete several lines of code of its context randomly to simulate real-world scenarios. EM and ES denotes the Exact Match and Edit Similarity score respectively. All results demonstrate that our Positional Integrity Encoding approach brings substantial speed-ups without performance drops.

	Model	Method	XF-F			XF-R			IF		
			EM	ES	Time	EM	ES	Time	EM	ES	Time
Python	1.3B	Full-recomputation	22.42	65.26	192ms	35.41	72.96	193ms	28.78	69.22	193ms
Python	1.3B	Conflict Fast Encoding	0.41	39.31	22ms	0.59	42.02	24ms	1.05	42.85	22ms
Python	1.3B	PIE	22.31	65.19	29ms	35.25	72.81	27ms	28.8	69.16	27ms
Python	6.7B	Full-recomputation	28.95	70.11	561ms	40.89	76.19	564ms	35.26	72.73	562ms
Python	6.7B	Conflict Fast Encoding	0.51	40.24	30ms	0.77	42.77	31ms	1.42	43.93	30ms
Python	6.7B	PIE	28.86	70.01	43ms	40.77	76.15	42ms	35.09	72.67	42ms
Python	33B	Full-recomputation	35.75	73.46	2194ms	46.0	78.9	2199ms	39.75	75.12	2194ms
Python	33B	Conflict Fast Encoding	1.42	43.93	105ms	1.55	44.22	108ms	2.4	45.04	108ms
Python	33B	PIE	35.09	72.67	128ms	45.21	78.18	118ms	39.69	75.05	119ms
Java	1.3B	Full-recomputation	26.21	70.89	200ms	36.77	76.31	200ms	45.89	78.04	198ms
Java	1.3B	Conflict Fast Encoding	0.33	33.48	22ms	0.55	36.24	22ms	1.35	38.65	22ms
Java	1.3B	PIE	26.05	70.77	28ms	36.83	76.34	27ms	45.74	77.91	27ms
Java	6.7B	Full-recomputation	32.51	75.56	578ms	41.97	79.41	578ms	50.86	80.53	578ms
Java	6.7B	Conflict Fast Encoding	0.49	33.6	29ms	0.8	36.17	29ms	1.84	38.82	29ms
Java	6.7B	PIE	32.57	75.56	42ms	41.83	79.39	43ms	50.88	80.52	42ms
Java	33B	Full-recomputation	35.05	76.93	2269ms	44.95	80.87	2281ms	53.23	81.76	2270ms
Java	33B	Conflict Fast Encoding	0.91	35.06	109ms	1.01	37.97	106ms	2.31	40.15	105ms
Java	33B	PIE	34.82	76.76	117ms	44.93	80.86	120ms	53.19	81.71	119ms

that all metrics indicate that our Positional Integrity Encoding can perfectly maintain the prediction accuracy of full re-computation in different scenarios. This indicates that PIE effectively addresses temporal confusion, and the semantic impact of minor edits is relatively negligible. For example, in the most challenging XF-F setting requiring to handle long-range cross-file context, the maximum relative difference between our PIE and Full-recomputation across different model sizes and code languages is 0.3%/0.15%, 0.66%/0.79%, 1.33%/2.24% for code insertion, deletion, and edition in

Table 4: **Performance comparisons of edition experiments.** In this task, for each next-line prediction target, we delete several lines of code of its context and simultaneously insert other lines of code randomly to simulate real-world scenarios. EM and ES denote the Exact Match and Edit Similarity score respectively. All results demonstrate that our Positional Integrity Encoding approach brings substantial speed-ups without performance drops.

	Model	Method	XF-F			XF-R			IF		
			EM	ES	Time	EM	ES	Time	EM	ES	Time
Python	1.3B	Full-recomputation	22.42	65.26	242ms	35.41	72.96	242ms	28.78	69.22	244ms
	1.3B	Conflict Fast Encoding	8.80	50.08	23ms	13.83	54.01	22ms	11.29	51.99	22ms
	1.3B	PIE	22.04	64.59	30ms	34.49	72.02	29ms	28.15	68.32	29ms
Python	6.7B	Full-recomputation	28.95	70.11	705ms	40.89	76.19	706ms	35.26	72.73	713ms
	6.7B	Conflict Fast Encoding	11.26	51.63	34ms	12.57	53.27	34ms	12.75	51.45	34ms
	6.7B	PIE	28.07	69.00	54ms	39.89	75.10	54ms	34.05	71.64	54ms
Python	33B	Full-recomputation	35.75	73.46	2766ms	46.00	78.90	2759ms	39.75	75.12	2787ms
	33B	Conflict Fast Encoding	14.33	53.73	126ms	15.12	54.59	121ms	13.88	51.94	127ms
	33B	PIE	34.62	72.47	146ms	44.59	77.69	142ms	38.83	74.08	141ms
Java	1.3B	Full-recomputation	26.21	70.89	251ms	36.77	76.31	253ms	45.89	78.04	249ms
	1.3B	Conflict Fast Encoding	5.87	31.46	22ms	8.01	32.74	23ms	10.20	34.34	23ms
	1.3B	PIE	25.29	68.93	30ms	35.59	74.17	30ms	44.51	76.06	29ms
Java	6.7B	Full-recomputation	32.51	75.56	733ms	41.97	79.41	736ms	50.86	80.53	728ms
	6.7B	Conflict Fast Encoding	9.28	39.32	34ms	11.47	39.11	35ms	15.51	41.92	33ms
	6.7B	PIE	31.32	73.83	52ms	40.89	77.65	53ms	49.44	78.80	53ms
Java	33B	Full-recomputation	35.05	76.93	2892ms	44.95	80.87	2894ms	53.23	81.76	2833ms
	33B	Conflict Fast Encoding	8.02	32.93	120ms	9.67	33.29	122ms	12.10	34.70	122ms
	33B	PIE	33.72	74.69	134ms	43.71	78.66	138ms	51.71	79.49	143ms

terms of EM/ES respectively, which is rather negligible. This thorough examination serves as a strong support of the reliability of our PIE approach in real-world code editing scenarios.

Positional Integrity Encoding significantly reduces computational overhead. Moreover, we further benchmark the computational costs brought by different approaches. From all results in the above tables, it can be easily seen that our Positional Integrity Encoding can achieve substantial speed-up compared to full re-computation while preserving its performance simultaneously. In particular, for the code edition experiment that requires both insertion and deletion, the averaged reductions of computational overhead induced by our PIE are 87.9%/88.2%, 92.4%/92.8%, 94.8%/95.2% for 1.3B, 6.7B, and 33B models on Python/Java languages respectively. Furthermore, compared to Conflict Fast Encoding which induces minimal costs but largely hurts performance, our PIE only brings negligible overhead, showing its good accuracy-efficiency balance.

In summary, our main results comprehensively demonstrate the superiority of our Positional Integration Encoding for code LLMs towards real-world code editing scenarios, which perfectly preserves the prediction accuracy and significantly addresses the crucial gap in the efficient deployment of LLMs in real-time dynamic scenarios.

4.3 More Analysis

In this subsection, we further present detailed analysis to investigate how large is the gap between our Positional Integrity Encoding and full re-computation in terms of context representations and predictions from LLMs, which provide additional insight of our approach.

How large is the gap on context representations? In practical scenarios, real-time editing by users results in the modified sequence $\mathbf{x}^{\text{edit}}$, requiring the KV cache to must be updated. In our analysis, we use the cosine similarity between context representations of full re-computation $\boldsymbol{K}^{*}_{[j+1:n]}$ and (1) our Positional Integrity Encoding $\boldsymbol{K}^{\text{edit}}_{[j+1:n]}$; (2) Conflict Fast Encoding $\boldsymbol{K}_{[j+1:n]}$. We employ the DeepSeek-Coder 6.7B model on the Python subset of RepoBench. Averaged results are reported.

In Figure 3, the cosine similarity between representations of full re-computation and our Positional Integraty Enocding is consistently around 1.0 across all layers. This high similarity demonstrates the

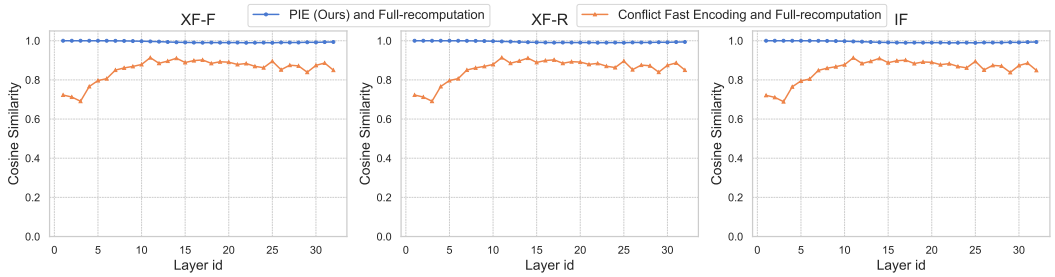

Figure 3: Cosine similarity of key representations across model layers. The plots compare the cosine similarity between $\boldsymbol{K}_{[j+1:n]}$ and $\boldsymbol{K}^*_{[j+1:n]}$ (indicating temporal confusion of Conflict Fast Encoding) with the cosine similarity between $\boldsymbol{K}^{\text{edit}}_{[j+1:n]}$ and $\boldsymbol{K}^*_{[j+1:n]}$ (showing the effectiveness of PIE).

effectiveness of PIE in preserving the contextual integrity of the key representations after editing, suggesting that PIE successfully mitigates the temporal confusion that typically arises when manipulating the KV cache. In contrast, the cosine similarity between representations of full re-computation and Conflict Fast Encoding is significantly lower, indicating the temporal confusion issue that hurts model performance a lot.

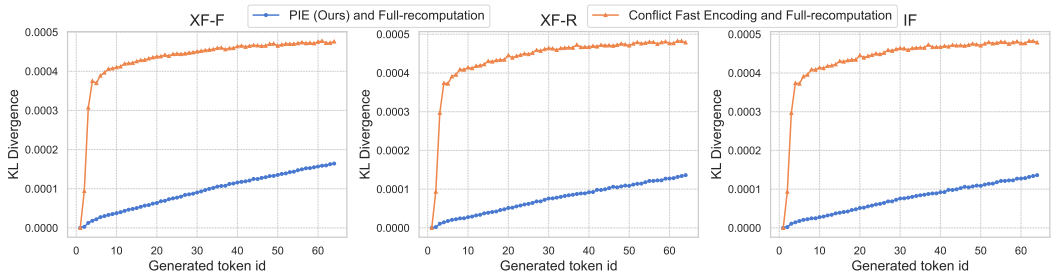

Figure 4: KL divergence of the generated token distributions. The plots compare the KL divergence between the generated token distributions of PIE and Full-recomputation, and the KL divergence between the generated token distributions of Conflict Fast Encoding and Full-recomputation.

How large is the gap on model predictions? Moreover, we further use Kullback-Leibler (KL) divergence as the metric to investigate the gap between model predictions of different approaches. Similarly, we employ the DeepSeek-Coder 6.7B model on the Python subset of RepoBench and report the averaged results. In Figure 4, the KL divergence between model predictions of full re-computation and our Positional Integrity Encoding remains consistently low, i.e., below 0.0002 across 64 tokens. However, the KL divergence between model predictions of full re-computation and Conflict Fast Encoding is substantially higher (2x larger). These findings again underscore the importance of maintaining positional integrity within the KV cache to ensure accurate generation results.

5 Conclusion

In this paper, we introduce Positional Integrity Encoding (PIE), a novel method designed to enhance the efficiency of large language models (LLMs) in the real-time editing setting. Our approach addresses the significant computational overhead associated with re-encoding contexts after small edits, a common scenario in interactive coding environments. Through extensive experiments, we demonstrated that PIE not only significantly reduces latency but also maintains high accuracy compared to the naive full re-computation method.

PIE represents a substantial step forward in the development of efficient LLMs, particularly in dynamic contexts where frequent edits are made. Future work could explore the integration of PIE with other optimization techniques and its application to a broader range of tasks beyond code generation. Our method paves the way for more responsive and resource-efficient AI assistants, enhancing their practicality and usability in various real-world scenarios.

REFERENCES

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint arXiv:2403.02310*, 2024.

Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv:2004.05150*, 2020.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023. URL https://arxiv.org/abs/2302.01318.

Ta-Chung Chi, Ting-Han Fan, Peter J Ramadge, and Alexander Rudnicky. Kerple: Kernelized relative positional embedding for length extrapolation. *Advances in Neural Information Processing Systems*, 35:8386–8399, 2022.

Ta-Chung Chi, Ting-Han Fan, Alexander Rudnicky, and Peter Ramadge. Dissecting transformer length extrapolation via the lens of receptive field analysis. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13522–13537, 2023.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. In *International Conference on Learning Representations*, 2021.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. In *acl*, July 2019.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023. URL https://arxiv.org/abs/2307.08691.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL https://arxiv.org/abs/2208.07339.

Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*, pp. 10323–10337. PMLR, 2023.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Optq: Accurate quantization for generative pre-trained transformers. In *The Eleventh International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=tcbBPnfwxS&fbclid=IwAR0QoRRF_7gr6NEt2sKchK5wgGNLfJUNavvbSeCRlWhpVmtUbo0W3ExJXZE.

Yao Fu. Challenges in deploying long-context transformers: A theoretical peak performance analysis. *arXiv preprint arXiv:2405.08944*, 2024.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. Minillm: Knowledge distillation of large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Zhenyu He, Zexuan Zhong, Tianle Cai, Jason D Lee, and Di He. Rest: Retrieval-based speculative decoding. *arXiv preprint arXiv:2311.08252*, 2023.

Zhenyu He, Guhao Feng, Shengjie Luo, Kai Yang, Di He, Jingjing Xu, Zhi Zhang, Hongxia Yang, and Liwei Wang. Two stones hit one bird: Bilevel positional encoding for better length extrapolation. *arXiv preprint arXiv:2401.16421*, 2024.

Itay Hubara, Brian Chmiel, Moshe Island, Ron Banner, Joseph Naor, and Daniel Soudry. Accelerated sparse neural training: A provable and efficient method to find n: m transposable masks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021. URL https://arxiv.org/abs/2102.08124.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pp. 5156–5165. PMLR, 2020.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Symposium on Operating Systems Principles (SOSP)*, 2023. URL https://arxiv.org/abs/2309.06180.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *International Conference on Machine Learning (ICML)*, 2023. URL https://proceedings.mlr.press/v202/leviathan23a.html.

Shanda Li, Chong You, Guru Guruganesh, Joshua Ainslie, Santiago Ontanon, Manzil Zaheer, Sumit Sanghai, Yiming Yang, Sanjiv Kumar, and Srinadh Bhojanapalli. Functional interpolation for relative positions improves long context transformers. *arXiv preprint arXiv:2310.04418*, 2023.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle: Speculative sampling requires rethinking feature uncertainty. In *International Conference on Machine Learning*, 2024.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems, 2024a. URL https://arxiv.org/abs/2306.03091.

Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 10012–10022, 2021.

Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. Llm-qat: Data-free quantization aware training for large language models. *arXiv preprint arXiv:2305.17888*, 2023. URL https://arxiv.org/abs/2305.17888.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for llm kv cache compression at test time. *Advances in Neural Information Processing Systems*, 36, 2024b.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Shengjie Luo, Shanda Li, Tianle Cai, Di He, Dinglan Peng, Shuxin Zheng, Guolin Ke, Liwei Wang, and Tie-Yan Liu. Stable, fast and accurate: Kernelized attention with relative positional encoding. *Advances in Neural Information Processing Systems*, 34:22795–22807, 2021.

Shengjie Luo, Shanda Li, Shuxin Zheng, Tie-Yan Liu, Liwei Wang, and Di He. Your transformer may not be as powerful as you expect. *Advances in Neural Information Processing Systems*, 35: 4301–4315, 2022.

Shengjie Luo, Tianlang Chen, Yixian Xu, Shuxin Zheng, Tie-Yan Liu, Liwei Wang, and Di He. One transformer can understand both 2d & 3d molecular data. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=vZTp1oPV3PC.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. Llm-pruner: On the structural pruning of large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. URL https://arxiv.org/pdf/2305.11627.pdf.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*, 2023. URL https://arxiv.org/abs/2305.09781.

Gunho Park, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. nuqmm: Quantized matmul for efficient inference of large-scale generative language models. *arXiv preprint arXiv:2206.09557*, 2022. URL https://arxiv.org/abs/2206.09557.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5:606–624, 2023.

Ofir Press, Noah Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations (ICLR)*, 2022.

Jiezhong Qiu, Hao Ma, Omer Levy, Wen-tau Yih, Sinong Wang, and Jie Tang. Blockwise self-attention for long document understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 2555–2565, 2020.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019. URL https://arxiv.org/abs/1910.01108.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Association for Computational Linguistics (ACL)*, June 2018.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pp. 31094–31116. PMLR, 2023.

Benjamin Spector and Chris Re. Accelerating llm inference with staged speculative decoding. *arXiv preprint arXiv:2308.04623*, 2023. URL https://arxiv.org/abs/2308.04623.

Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. Blockwise parallel decoding for deep autoregressive models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/c4127b9194fe8562c64dc0f5bf2c93bc-Paper.pdf.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2021.

Yutao Sun, Li Dong, Barun Patra, Shuming Ma, Shaohan Huang, Alon Benhaim, Vishrav Chaudhary, Xia Song, and Furu Wei. A length-extrapolatable transformer. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, July 2023.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pp. 1433–1443, 2020.

Yi Tay, Dara Bahri, Liu Yang, Donald Metzler, and Da-Cheng Juan. Sparse sinkhorn attention. In *International Conference on Machine Learning*, pp. 9438–9447. PMLR, 2020.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021. URL https://arxiv.org/abs/2012.09852.

Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning (ICML)*, 2023. URL https://proceedings.mlr.press/v202/xiao23c/xiao23c.pdf.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=NG7sS51zVF.

Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/adf7fa39d65e2983d724ff7da57f00ac-Paper-Conference.pdf.

Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform badly for graph representation? *Advances in neural information processing systems*, 34:28877–28888, 2021.

Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, Weng Lam Tam, Zixuan Ma, Yufei Xue, Jidong Zhai, Wenguang Chen, Zhiyuan Liu, Peng Zhang, Yuxiao Dong, and Jie Tang. GLM-130b: An open bilingual pre-trained model. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=-Aw0rrrPUF.

Bohang Zhang, Shengjie Luo, Liwei Wang, and Di He. Rethinking the expressive power of GNNs via graph biconnectivity. In *The Eleventh International Conference on Learning Representations*, 2023a. URL https://openreview.net/forum?id=r9hNv76KoT3.

Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. Draft & verify: Lossless large language model acceleration via self-speculative decoding. *arXiv preprint arXiv:2309.08168*, 2023b.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

A Experimental Details

A.1 Experimental Setup

Tasks Construction for Code Insertion. To simulate code insertion tasks, we start by randomly deleting five consecutive lines from each context. The resulting context, which lacks these five lines, is considered the original context. The complete context, which includes the previously deleted lines, is treated as the edited context. The tokens within the deleted lines are identified as the inserted tokens (around 64 tokens for Python and 51 tokens for Java). This setup allows us to evaluate the model's capability to accurately restore missing code segments, mimicking real-world scenarios where developers frequently insert blocks of code.

Tasks Construction for Code Deletion. For code deletion tasks, we begin by randomly selecting a line within the context and then inserting five randomly sampled lines at this position. The context containing these additional lines is designated as the original context. The complete context, which excludes the inserted lines, is regarded as the edited context. The tokens in the inserted lines are treated as the deleted tokens (around 64 tokens for Python and 51 tokens for Java). This construction enables us to assess the model's performance in identifying and removing extraneous code, reflecting situations where developers need to clean up or refactor their codebase.

Tasks Construction for Multi-place Code Edition To comprehensively evaluate the model's performance in handling simultaneous code insertion and deletion, we construct a task scenario that integrates both operations. Initially, we randomly delete five consecutive lines from each context to simulate code insertion. The context without these lines is treated as the original context. The tokens in the deleted lines are identified as the inserted tokens.

Simultaneously, we randomly select another line within the context and insert five randomly sampled lines at this position. The complete context, which includes all lines as they appear after both deletions and insertions, is regarded as the edited context. The tokens in the newly inserted lines are considered the deleted tokens. This dual operation setup allows us to evaluate the model's ability to handle complex, simultaneous edits, adding missing code segments while removing extraneous ones, reflecting the multifaceted nature of real-world coding environments where developers often perform multiple types of edits concurrently.

A.2 Results on CodeLlama

Results of different code editing settings on CodeLlama (Roziere et al., 2023) are presented in Table 5, 6 and 7 respectively. Similar to DeepSeek-Coder, CodeLlama with Positional Integrity Encoding demonstrates strong performance and fast speed across various editing scenarios. The metrics indicate that PIE effectively maintains prediction accuracy and addresses temporal confusion, ensuring the impact of minor edits is minimal.

Table 5: **Performance comparisons of insertion experiments for CodeLlama.** In this task, for each next-line prediction target, we insert several lines of code into its context randomly to simulate real-world scenarios. EM and ES denotes the Exact Match and Edit Similarity score respectively. All results demonstrate that our Positional Integrity Encoding approach brings substantial speed-ups without performance drops.

	Model	Method	XF-F			XF-R			IF		
			EM	ES	Time	EM	ES	Time	EM	ES	Time
Python	7B	Full-recomputation	25.72	66.49	561ms	38.87	73.81	564ms	33.55	70.26	562ms
	7B	Conflict Fast Encoding	14.49	55.10	34ms	19.32	57.34	34ms	15.49	53.05	34ms
	7B	PIE	25.42	66.40	50ms	38.76	73.83	50ms	33.30	70.11	50ms
Python	34B	Full-recomputation	31.04	69.48	2013ms	42.80	76.27	2029ms	37.69	72.36	1999ms
	34B	Conflict Fast Encoding	11.56	50.23	113ms	16.06	52.59	110ms	11.58	48.36	112ms
	34B	PIE	30.53	68.79	123ms	42.73	76.40	119ms	37.51	72.25	123ms
Java	7B	Full-recomputation	28.02	72.49	578ms	39.61	77.81	578ms	49.20	79.76	578ms
	7B	Conflict Fast Encoding	10.45	37.69	34ms	14.28	38.45	35ms	16.95	38.15	33ms
	7B	PIE	28.02	72.39	50ms	39.48	77.74	49ms	49.31	79.79	48ms
Java	34B	Full-recomputation	32.16	75.09	2179ms	43.45	79.89	2184ms	52.43	80.97	2197ms
	34B	Conflict Fast Encoding	12.04	41.76	104ms	14.25	39.19	107ms	17.29	38.50	109ms
	34B	PIE	31.45	74.64	119ms	43.54	79.80	118ms	52.42	81.00	115ms

Table 6: **Performance comparisons of deletion experiments for CodeLlama.** In this task, for each next-line prediction target, we delete several lines of code of its context randomly to simulate real-world scenarios. EM and ES denote the Exact Match and Edit Similarity score respectively. All results demonstrate that our Positional Integrity Encoding approach brings substantial speed-ups without performance drops.

	Model	Method	XF-F			XF-R			IF		
			EM	ES	Time	EM	ES	Time	EM	ES	Time
Python	7B	Full-recomputation	25.72	66.49	561ms	38.87	73.81	564ms	33.55	70.26	562ms
	7B	Conflict Fast Encoding	10.14	52.80	30ms	15.24	57.88	31ms	13.04	54.53	30ms
	7B	PIE	25.52	66.47	43ms	38.88	73.87	42ms	33.51	70.18	42ms
Python	34B	Full-recomputation	31.04	69.48	2013ms	42.80	76.27	2029ms	37.69	72.36	1999ms
	34B	Conflict Fast Encoding	12.15	54.76	88ms	16.40	59.29	94ms	13.78	55.90	93ms
	34B	PIE	31.01	69.44	100ms	42.76	76.13	100ms	37.60	72.38	102ms
Java	7B	Full-recomputation	28.02	72.49	578ms	39.61	77.81	578ms	49.20	79.76	578ms
	7B	Conflict Fast Encoding	7.21	42.51	29ms	7.77	43.86	29ms	8.78	44.46	29ms
	7B	PIE	28.01	72.45	42ms	39.51	77.78	42ms	49.30	79.80	42ms
Java	34B	Full-recomputation	32.16	75.09	2179ms	43.45	79.89	2184ms	52.43	80.97	2197ms
	34B	Conflict Fast Encoding	8.76	45.44	92ms	8.67	45.57	92ms	10.11	46.64	95ms
	34B	PIE	32.12	75.02	99ms	43.33	79.86	98ms	52.36	80.96	107ms

Table 7: **Performance comparisons of edition experiments for CodeLlama.** In this task, for each next-line prediction target, we delete several lines of code of its context and simultaneously insert other lines of code randomly to simulate real-world scenarios. EM and ES denote the Exact Match and Edit Similarity score respectively. All results demonstrate that our Positional Integrity Encoding approach brings substantial speed-ups without performance drops.

	Model	Method	XF-F			XF-R			IF		
			EM	ES	Time	EM	ES	Time	EM	ES	Time
Python	7B	Full-recomputation	25.72	66.49	705ms	38.87	73.81	706ms	33.55	70.26	713ms
	7B	Conflict Fast Encoding	19.97	61.46	34ms	29.79	67.31	34ms	24.70	63.28	34ms
	7B	PIE	24.97	66.04	54ms	38.21	73.40	54ms	32.84	69.66	54ms
Python	34B	Full-recomputation	31.04	69.48	2574ms	42.80	76.27	2569ms	37.69	72.36	2581ms
	34B	Conflict Fast Encoding	21.91	61.78	91ms	30.87	67.16	91ms	25.76	63.15	90ms
	34B	PIE	29.92	68.7	126ms	41.74	75.37	126ms	37.01	71.76	127ms
Java	7B	Full-recomputation	28.02	72.49	733ms	39.61	77.81	736ms	49.20	79.76	728ms
	7B	Conflict Fast Encoding	18.78	57.38	34ms	25.79	60.97	35ms	32.60	62.66	33ms
	7B	PIE	27.43	71.60	52ms	38.89	76.96	53ms	48.60	78.94	53ms
Java	34B	Full-recomputation	32.16	75.09	2726ms	43.45	79.89	2807ms	52.43	80.97	2727ms
	34B	Conflict Fast Encoding	21.25	59.70	104ms	27.83	61.91	107ms	32.98	62.03	109ms
	34B	PIE	30.82	73.84	126ms	42.71	78.94	119ms	51.64	80.15	125ms