EFIM: Efficient Serving of LLMs for Infilling Tasks with Improved KV Cache Reuse

Tianyu Guo[1*†][0009−0005−2979−4486], Hande Dong[2*✉], Yichong Leng[3], Feng Liu[2], Cheater Lin[2], Nong Xiao[1][0000−0002−2166−977X], and Xianwei Zhang[1✉][0000−0003−3507−4299]

[1] Sun Yat-sen University, Guangzhou, China
guoty9@mail2.sysu.edu.cn,{xiaon6,zhangxw79}@mail.sysu.edu.cn
[2] Tencent, Shenzhen, China
{donghd66,neolscarlet}@gmail.com,cheaterlin@tencent.com
[3] University of Science and Technology of China, Hefei, China
lyc123go@mail.ustc.edu.cn

Abstract. Large language models (LLMs) are often used for infilling tasks, which involve predicting or generating missing information in a given text. These tasks typically require multiple interactions with similar context. To reduce the computation of repeated historical tokens, cross-request key-value (KV) cache reuse, a technique that stores and reuses intermediate computations, has become a crucial method in multi-round interactive services. However, in infilling tasks, the KV cache reuse is often hindered by the structure of the prompt format, which typically consists of a prefix and suffix relative to the insertion point. Specifically, the KV cache of the prefix or suffix part is frequently invalidated as the other part (suffix or prefix) is incrementally generated. To address the issue, we propose EFIM, a transformed prompt format of FIM to unleash the performance potential of KV cache reuse. Although the transformed prompt can solve the inefficiency, it exposes subtoken generation problems in current LLMs, where they have difficulty generating partial words accurately. Therefore, we introduce a fragment tokenization training method which splits text into multiple fragments before tokenization during data processing. Experiments on two representative LLMs show that LLM serving with EFIM can lower the latency by 52% and improve the throughput by 98% while maintaining the original infilling capability. EFIM's source code is publicly available at https://github.com/gty111/EFIM.

Keywords: FIM · KV cache · Subtoken · LLM serving.

1 Introduction

Infilling tasks involve predicting or generating missing words, phrases, or even entire sentences within a given text. Recently, there has been a growing trend of

* Equal contribution.
† This work was done during an internship at Tencent.

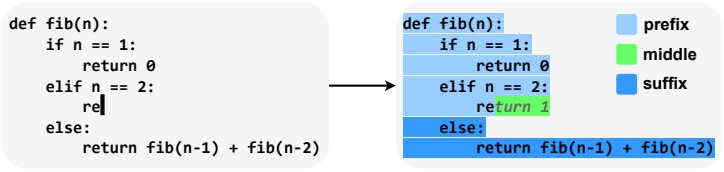


Fig. 1: A python code snippet where a programmer wants to insert code inside a function. The prefix/suffix part represents content before/after the insertion point. The middle part is the content expected to infill.

using large language models (LLMs) like Codex [6], StarCoder [25, 27], CodeLlama [31], Qwen2.5-coder [22] and DeepSeek-Coder [20] for such tasks. As a result, many companies are starting to provide online services for infilling, such as OpenAI canvas [5], GitHub Copilot [9] and Amazon CodeWhisperer [8]. However, prompts of infilling tasks require long context around the insertion point and users often need multi-turn interactions with LLMs, leading to high computational demands. Efficient serving of LLMs for infilling tasks has become an important research problem (a detailed analysis is given in §5.3).

As the *de facto* technique to reduce computation and accelerate LLMs inference, KV cache [1, 23, 30] stores attention keys and values to prevent recomputation. While traditional KV cache operates within a single request, cross-request KV cache reuse[1] [18, 37, 39, 41] has been proposed to minimize redundant KV cache recomputation in multi-turn services [34], significantly reducing latency. However, cross-request KV cache reuse imposes strict constrains that prefix of prompt tokens must remain identical. In the infilling scenario as shown in Figure 1, the prompt typically follows the fill-in-the-middle (FIM) format [3], i.e., "<P>prefix<S>suffix<M>", where <P>, <S> and <M> are FIM special tokens to connect prefix, suffix and middle parts. A common behavior in infilling tasks is the continuous expansion of the prefix, which invalidates the KV cache of the suffix. This occurs because incremental changes in the prefix alter the preceding tokens of the suffix. Thus, KV cache of the suffix need to be recomputed in each interaction, requiring high computation resources. To solve it, we propose transforming the FIM format from "<P>prefix+*inc*<S>suffix<M>" to "<P>prefix<S>suffix<M>*inc*" (EFIM), where *inc* represents the incremental prefix change. This modification ensures that both the prefix and suffix remain unchanged, with the variation confined to *inc*. Consequently, KV cache reuse can be extended from solely the prefix to include both the prefix and suffix.

Despite EFIM improves KV cache reuse, it reveals a hidden subtoken[2] generation problem in current LLMs. The issue stems from EFIM's requirement for models to generate subtokens after *inc*, a capability not supported by existing LLMs. To enable universal subtoken generation, we propose a fragment tokenization training method, involving randomly splitting sentences into multiple segments, tokenizing each segment individually, and then concatenating the re-

[1] If not explicitly stated, KV cache reuse in this paper is cross-request.

[2] In the paper, we refer to incomplete words as subtokens like "pri" in "print". Incomplete words caused by tokenizer are not included.

sults. In this way, the model can learn the ability to generate the remaining subtokens based on the initial subtoken during training, thereby addressing the subtoken issue encountered by EFIM.

In summary, the contributions of this paper are:

- We identify that the efficiency of LLM inference for infilling tasks is hindered by the FIM format, as the KV cache of the prefix/suffix part is frequently invalidated by the growing suffix/prefix.
- We propose EFIM, the first method to transform the FIM prompt format, unlocking the potential of KV cache reuse.
- To enhance subtoken generation ability, we introduce a fragment tokenization training method on data processing.
- Experiments on two pretrained LLMs show that EFIM reduces average latency by 52% and increases throughput by 98%, while preserving model capability.

2 BACKGROUND AND MOTIVATION

2.1 Training LLMs with FIM

Fig. 2: Comparison of PSM and SPM. <P> follows prefix part, <S> follows suffix part, <M> follows middle part and <E> marks the end of infilling span.

Current decoder-based autoregressive (AR) language models [2, 4, 7, 14] are capable of generating text from left to right. However, they struggle with infilling tasks, where the model is required to generate text at a specific location within a snippet, conditioned on both a prefix and a suffix. To address this limitation, FIM capabilities have been integrated into AR models without compromising their standard left-to-right generation [3, 17, 28, 29]. The core idea of FIM involves splitting the documents into three parts, and then relocating the middle part to the end. Models are trained on a mixture of FIM transformed data and standard left-to-right data. As shown in Figure 2, FIM can be prepared in two ways denoted as prefix-suffix-middle (PSM) and suffix-prefix-middle (SPM). In general, the LLMs can own both abilities.

2.2 KV Cache Reuse Inefficiency with FIM

LLMs notably feature their self-attention mechanism, and the KV cache is used to accelerate the inference [10, 11, 24, 33, 35, 36, 38, 40]. Additionally, the KV cache of shared prefix across different sequences can be reused to avoid redundant computations [18, 19, 37, 39]. In infilling scenarios, users usually need to engage in

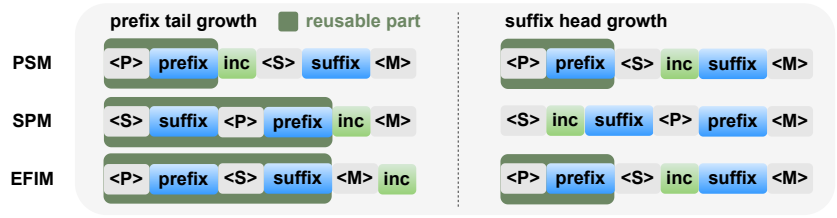
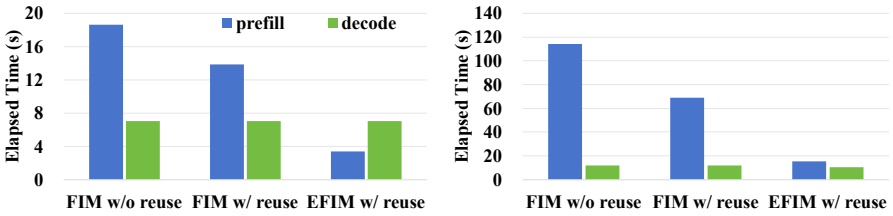


Fig. 3: Reusable part between PSM, SPM and EFIM when the growth (*inc*) happens either at the prefix tail or at the suffix head. The reusable part includes the content before *inc*.

multi-round interactions with LLMs, especially when dealing with long contexts. According to our statistics from online infilling services, most of modifying behaviors involve appending tokens to the tail of the prefix or the head of the suffix. Figure 3 illustrates the reusable parts of the KV cache across different prompt formats. It shows that changes to the tail of the prefix invalidate the KV cache of the suffix in PSM, while changes to the head of the suffix invalidate both the prefix and suffix in SPM. To address unnecessary KV cache invalidation, we propose EFIM, which relocates the prefix increment to the end of the prompt in PSM. EFIM combines the advantage of PSM and SPM, achieving the most KV cache reuse in both scenes (prefix tail growth and suffix head growth).



(a) Deepseek-coder-6.7B (b) Llama3.1-8B

Fig. 4: Elapsed time breakdown of prefill and decode stage for infilling serving (average input/output length is 2100/32) between FIM and EFIM.

We also analyze the end to end elapsed time breakdown for infilling serving as illustrated in Figure 4. Without cross-request KV cache reuse (FIM w/o reuse), the prefill overhead can be up to 9 times (114 vs. 12) than decode. Even with KV cache reuse (FIM w/ reuse), the time gap between prefill and decode remains significant, reaching up to 6 times (69 vs. 12). With EFIM, the overhead of prefill stage is significantly reduced by 40% on average (114 vs. 69) compared to FIM w/ reuse. This demonstrates EFIM's superior computational efficiency during the prefill phase.

2.3 Subtoken Generation Capability with LLMs

During the pre-training of LLMs, the models are typically trained on vast corpora of text data to assimilate the statistical regularities and semantic repre-

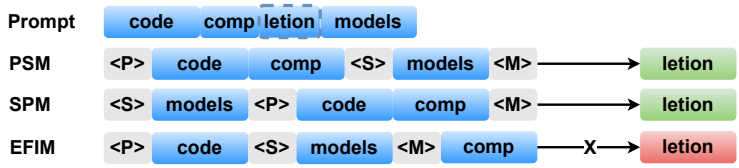


Fig. 5: Subtoken generation ability between different prompt formats considering prompt "code comp[] models".

sentations of language. Despite their prowess in generating coherent text, LLMs exhibit limitations when it comes to handling subtoken generation tasks due to the lack of relevant cases in their training data. For instance, existing models fail to generate "nt" after "pri". Whereas infilling LLMs overcome this limitation by training on documents split into three parts and joined with FIM special tokens. This process introduces subtokens into the dataset, as the splits created by the FIM special tokens often result in partial tokens (subtokens). Therefore, subtokens typically appear around FIM special tokens, i.e., "subtoken<M>subtoken", and their generation relies on the context provided by these tokens. Without this context, the model loses the ability to generate subtokens effectively. For example, as shown in Figure 5, when the input prompt is 'code comp[] models', where '[]' represents the missing content, both PSM and SPM can successfully generate the subtoken 'letion'. However, EFIM fails to generate sutokens correctly when the prefix ends with a subtoken, highlighting the limitations of directly applying LLMs in such cases. To address this challenge, we must enhance LLMs with a universal subtoken generation capability, ensuring that the model can generate subtokens regardless of the presence of FIM special tokens.

3 DESIGN

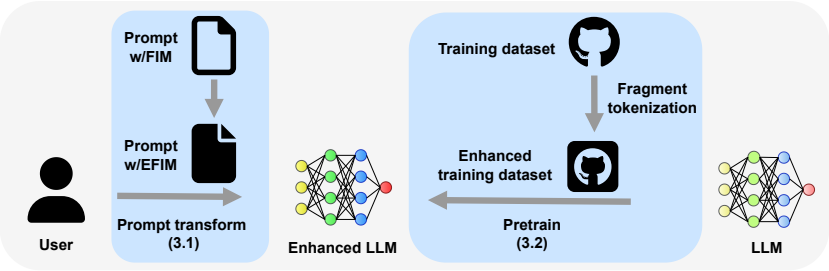


Fig. 6: Overall diagram of design with EFIM.

Our proposed design with EFIM consists of two key parts as illustrated in Figure 6. The first part operates between the user and the LLM to seamlessly and automatically convert the prompt format from FIM to EFIM. This transformation is fully transparent to the user, ensuring a smooth and intuitive experience. The second part introduces a fragment tokenization training method focused on

data processing. This method is designed to augment the LLM's ability to generate subtokens, a critical requirement for EFIM functionality. Our implementation introduces no architectural changes, making EFIM accessible for integration into existing LLM frameworks.

3.1 From FIM to EFIM

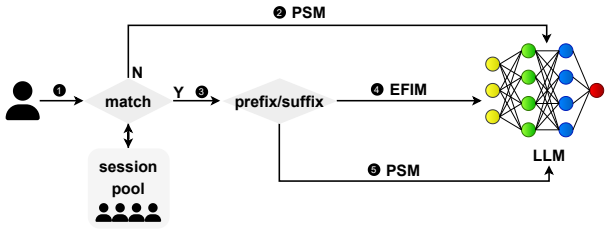


Fig. 7: The prompt transformation process from FIM to EFIM.

To automatically convert prompt format from FIM to EFIM, we use a per-user session pool to track the most recent interaction between users and the LLMs as shown in Figure 7. Each session stores the prefix and suffix parts extracted from the user's previous request. ❶ When a new request is received, we first check if the user has an existing session in the pool and identify the prefix and suffix parts. ❷ If no matching session is found, we forward the prompt in PSM format to the LLM inference engine and create a new session for the user. ❸ If a matching session is located, we compare the prefix/suffix in the new request with the one from the previous interaction. ❹ If the prefix in the new request contains additional content compared to the session prefix, we split the new prefix into a common part and an incremental part referred to as *inc*. We then construct the EFIM-formatted prompt by concatenating the common part, the new suffix, and *inc*, before sending it to the LLM. In this way, the incremental prefix content does not invalidate the KV cache for the suffix, unlike in the PSM format. ❺ If the suffix of new request has an incremental part compared to the session suffix, we send the request in PSM format directly to the LLM inference engine. In this scenario, the KV cache for the common prefix can still be reused, offering an advantage over the SPM format.

3.2 Fragment Tokenization Training Method

To equip LLMs with universal subtoken generation capability, we propose a novel fragment tokenization training method focused on data processing. It fundamentally differs from FIM in how the training dataset is processed. Figure 8 shows the similarities and differences parts between the two approaches. Both FIM and our method apply the transformation to the documents to adjust the order of prefix, suffix and middle. While FIM directly tokenize the three parts, our method split the text into multiple segments to allow subtokens to be generated at more locations. We also provide an example on data processing in Figure 9.

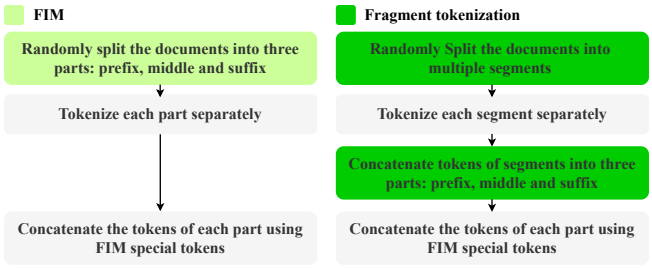


Fig. 8: Data processing diagram between FIM (left) and fragment tokenization (right). The length of each segment follows uniform distribution [1,200].

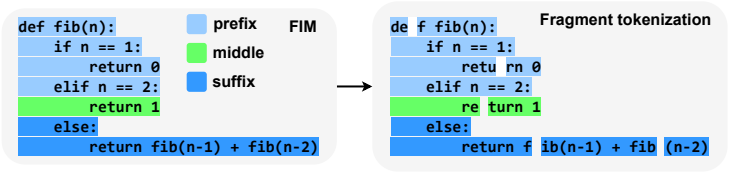


Fig. 9: Comparison between FIM (left) and fragment tokenization (right) data processing examples. FIM-based method only splits the text into three parts, while our method splits the text into multiple segments and employ tokenization for each segment.

The fragment tokenization approach allows subtokens to appear not only adjacent to FIM special tokens but also throughout any position in the sequence. As a result, the model develops a more comprehensive and universal subtoken generation capability. By embedding subtokens across varied contexts within the training data, the enhanced LLM becomes better equipped to generate subtokens seamlessly in diverse scenarios, making it far more versatile and effective for real-world applications. *It is important to note that our approach can serve as a drop-in replacement of current LLM training process, incurring no additional overhead. For existing LLMs, our method can be applied during continued pretraining.*

4 EXPERIMENTAL METHODOLOGY

We conduct continue pretraining with 64 A100 GPUs on two representative LLMs, Deepseek-coder-6.7B[3] [20] and Llama3.1-8B [16], using fragment tokenization method to enhance their sub-token generation ability. The pretraining process for each model takes less than a week. Notably, the additional overhead can be avoided if the fragment tokenization training method is applied from the beginning. The training dataset consists of 108 billion tokens collected from StarCoderData [32]. For Llama3.1-8B, we pretrain a baseline version (based on the original LLM) to equip it with FIM ability. The experiments mainly focus on three questions:

[3] This enhanced model has been used in production for AI Code Assistant.

1. Does fragment tokenization method impact infilling ability and truely make LLMs possess subtoken generation ability? (§4.1 and §5.1)
2. Can EFIM improve the KV cache reuse and the efficiency of LLM serving? (§4.2 and §5.2)
3. Is it worth the training overhead to gain inference speed? (§5.3)

4.1 Infilling and Subtoken Generation Ability

Fig. 10: Prompt creation procedure and prompt format between FIM and EFIM.

Current infilling evaluations rely on FIM, which is incompatible with EFIM. To assess both infilling and universal subtoken abilities with EFIM, we adapt the prompt format from HumanEval Infilling [3] and CrossCodeEval (CCEval) [15], focusing on the scenario where tokens are appended to the prefix. This scenario highlights the behavioral differences between PSM and EFIM.

Prompt creation. Figure 10 illustrates the prompt creation process and prompt format of FIM and EFIM. Based on HumanEval Infilling, we randomly split the prefix into prefix1 and prefix2, use prefix1 as the original prefix and prefix2 as the increment of prefix. Note that, HumanEval Infilling includes three infiling benchmarks, single-line, multi-line and random-span. In the single-line and multi-line benchmarks, prefix2 does not end with subtokens because they require the generation of complete single or multiple lines of code. In contrast, the random-span benchmark may have subtokens at the end of prefix2. In CCEval, we modify the prompt format by randomly splitting the entire code snippet into four parts (prefix1, prefix2, content to infill and suffix). Prefix1 is used as the original prefix, suffix as the original suffix and prefix2 as the increment of prefix. Since the splitting process is entirely random, prefix2 may end with subtokens.

Metrics We use pass@1 for HumanEval Infilling, EM and ES for CCEval.

- **Pass@1**: One code sample is generated per problem, a problem is considered solved if the sample passes the unit tests, and the percent of problems solved is reported.
- **Exact Match (EM)**: The percent of the situations when the generated code is exact the ground truth.
- **Edit Similarity (ES)**: Similarity score between the generated code and the ground truth using the Levenshtein distance algorithm. The score ranges from 0 to 100, where higher values indicate greater similarity.

Schemes We conduct a comparative analysis between the original LLM (oLLM) and our proposed enhanced LLM (eLLM), both of which utilize FIM or EFIM.

4.2 Inference Speedup

To evaluate the efficiency of different reusable KV cache levels (none, prefix, and prefix+suffix), we compare EFIM with FIM[4] in a scenario where tokens are appended to the prefix. This setup simulates infilling cloud services, where multiple users interact with LLMs over several rounds. In each round, a prefix is extended with new tokens. Instead of a fixed request rate, an unrealistic scenario, we adjust the service load based on the number of users. Each user acts as an individual client, sending a request for the next round only after receiving the previous response. For our experiment, we set the number of rounds to 5 and the number of users to 16. The average input/output length is 2355/128.

Environment We utilize the vLLM inference framework (v0.6.2) [23]. The experiments are performed on a server with an AMD EPYC 7742 processor, 256GB of host memory and an NVIDIA A100 GPU.

Metrics

- **Latency**: Average end to end latency for each request.
- **Input throughput**: Average input token processing throughput.
- **Request throughput**: Average request completion rate.
- **Reuse rate**: Cross-request KV cache reuse rate.

Schemes

- **Baseline**: PSM without KV cache reuse.
- **FIM**: PSM with KV cache reuse.
- **EFIM**: EFIM with KV cache reuse.

5 RESULTS AND ANALYSIS

5.1 Infilling and Subtoken Generation Ability

Table 1 presents the evaluation results of infilling performance. For the HumanEval Infilling single/multi-line benchmark, the pass rates between *oLLM w/FIM* and *oLLM w/EFIM* remain close (with a difference of less than 1%) as the single/multi-line tasks do not require subtoken generation (there are no subtokens at the end of the prefix increment). This demonstrates that EFIM has little influence when subtoken generation is not required. However, for the random-span benchmark, the pass rate drops significantly by 24% from *oLLM w/FIM* to *oLLM w/EFIM*, highlighting the model's inability to generate subtokens. In contrast, *eLLM w/EFIM* can maintain equivalent performance compared to *oLLM*

[4] The advantage of EFIM compared to FIM can be seen different reusable KV cache levels. Therefore, in this experiment, we focus on the efficiency of serving at different reusable KV cache levels.

Table 1: Evaluation results on infilling benchmarks. The left part shows Pass@1 rate (higher is better) in HumanEval Infilling where S stands for single-line, M stands for multi-line and R stands for random-span. The right part shows EM and ES metric (higher is better) in CCEval. oLLM and eLLM abbreviate for LLM training with FIM and fragment tokenization, respectively. The underlined numbers indicate a decrease in the model's ability due to the lack of subtoken generation capability.

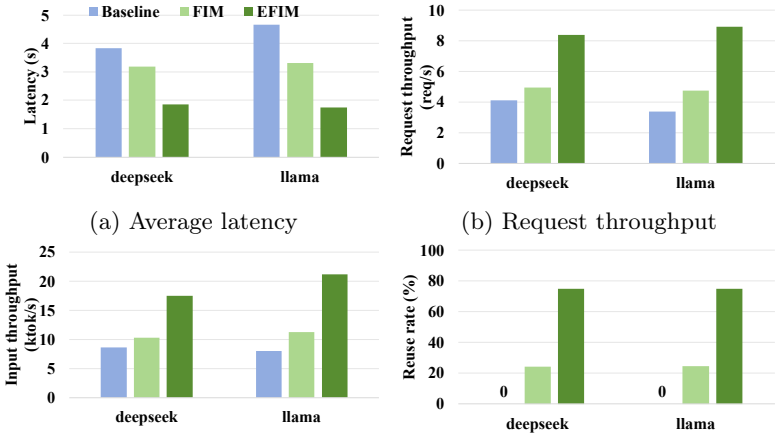
Benchmark	**HumanEval Infilling**						**CCEval**			
Model	**Deepseek**			**Llama**			**Deepseek**		**Llama**	
	S	M	R	S	M	R	EM	ES	EM	ES
oLLM w/FIM	89.64	61.96	76.77	87.32	56.90	62.99	33.51	78.43	29.40	71.30
oLLM w/EFIM	90.03	62.25	<u>52.44</u>	86.35	56.54	<u>38.35</u>	<u>11.19</u>	<u>71.04</u>	<u>6.82</u>	<u>53.44</u>
eLLM w/FIM	88.48	61.62	75.12	87.12	57.73	67.20	33.27	79.24	31.51	71.15
eLLM w/EFIM	89.64	62.82	75.61	86.83	56.35	64.27	32.51	78.91	30.91	70.48

w/FIM, indicating that the fragment tokenization method (§3.2) can effectively solve subtokens generation problems. *eLLM w/FIM* also exhibits close performance compared to *oLLM w/FIM*, showing that the fragment tokenization method has little impact on infilling ability. For CCEval, the metrics shows similar pattern. Compared to ES, EM shows a more significant decrease as the model struggles to generate subtokens but performs well in generating other types of content.

5.2 Inference Speedup

Figure 11 illustrates the overall inference performance. Among the three schemes, *Baseline* performs the worst due to the lack of KV cache reuse, requiring the entire prompt's KV cache to be recomputed in each round which is highly time consuming. Instead, *FIM* reduces latency by 21% and improves throughput by 26% on average by avoiding the recomputation of the prefix's KV cache. However, it still requires recomputing the suffix's KV cache due to the inefficiency of FIM. *EFIM* addresses this issue, achieving an average latency reduction of 52% and a throughput increase of 98%. Besides, the average latency per request drops below 2 seconds, significantly enhancing user experience. *EFIM* achieves the lowest latency and highest throughput by maximizing KV cache reuse, as evidenced by the highest input token throughput.

Number of concurrently serving users. To evaluate the impact of the number of concurrently serving users, we conduct a sensitivity study. Figure 12 illustrates the average latency and KV cache reuse rate as user count increases. From the results, we observe that the latency of *FIM* increases almost proportionally with the number of users. In contrast, *EFIM* exhibits a steeper latency curve as the user count grows, which can be attributed to a significant



(a) Average latency (b) Request throughput

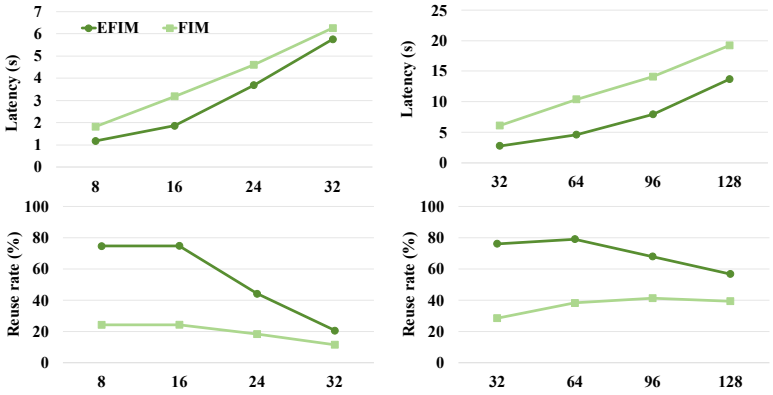
(c) Input token throughput (d) KV cache reuse rate

Fig. 11: Overall inference performance on average latency, request throughput, input token throughput and KV cache reuse rate to illustrate the efficiency of different degrees of reusable KV cache.

decline in its KV cache reuse rate. When the number of users is relatively low, *EFIM* maintains a stable reuse rate of around 80%. However, as the user count increases, the total capacity of the KV cache for completed requests gradually exceeds available GPU memory, leading to a drop in reuse rate. On the other hand, *FIM* consistently shows a lower reuse rate, remaining below 40% across all user counts.

5.3 Cost Efficiency

While existing LLMs require continued pretraining to enable subtoken generation abilities, our method demonstrates superior cost efficiency. For instance, Meta's Llama3.1-8B model requires 1.46 million H100 GPU hours for training [26]. In contrast, our fragment tokenization approach consumes only 10,752 A100 GPU hours ($64 \times 7 \times 24$), representing merely 0.74% of training cost of Llama3.1-8B. According to the Deepseek technical report [12, 13], the Deepseek V3 model requires 2.788 million H800 GPU hours for training, with an average daily serving cost of 43,536 H800 GPU hours per day (1.56% of its training cost). By improving throughput by 98%, EFIM reduces serving costs by 49.5% ($1 - \frac{1}{1+0.98}$), which translates to 0.77% of the total training cost. This reduction enables the training cost for fragment tokenization method to be offset within a single day. It is important to note that Deepseek is used here as an illustrative example. Other companies may incur higher serving costs depending on their specific deployment scenarios. Nevertheless, the cost efficiency of EFIM remains a compelling advantage for scaling LLM inference.



(a) deepseek (b) llama

Fig. 12: Variation of latency (above) and KV cache reuse rate (below) as the number of users (horizontal axis) increases.

6 RELATED WORK

Cross-request KV cache reuse. Cross-request KV cache reuse is a key feature in LLM inference framework [23,41], aimed at reducing computation during the prefill stage. Several studies [18,19,37,39] have addressed the challenge of limited GPU memory for storing KV cache by utilizing CPU host memory or even disk storage to expand capacity. While these approaches focus on leveraging physical resources to improve KV cache reuse, our work improves it by transforming the prompt format in the infilling scene.

LLMs for infilling tasks. Using LLMs to infill contents has become a crucial technique in assisted programming, with numerous open-source models developed to support this application [17,20,22,25,27–29,31]. Existing research typically focuses on acquiring, curating, and generating large-scale training datasets, as well as optimizing the training process to enhance the performance and accuracy of infilling tasks. In contrast, our work targets a specific aspect of model functionality which improves the subtoken generation ability without compromising overall model performance.

7 CONCLUSION

This paper identifies that the efficiency of LLM inference in infilling tasks can be hindered by the FIM format. To address this issue, we propose EFIM, a modified format designed to increase KV cache reuse. However, EFIM reveals universal subtoken generation problems in current LLMs. To solve it, we introduce an augmented training method during data processing to empower LLMs' sub-token generation. Experiments on two typical LLMs shows that EFIM reduces average latency by 52% and increases throughput by 98%, while maintaining the model's original capabilities.

Acknowledgements and Artifact Availability. We are grateful to the anonymous reviewers for their helpful suggestions. Special thanks are extended to Yi Liu and Qiang Lin at Tencent for their contributions. This research was supported by the National Natural Science Foundation of China-#62472462/#62402534/#62461146204, and sponsored by CCF-Tencent Rhino-Bird Open Research Fund (CCF-Tencent RAGR20240102). The artifact is available in the Zenodo repository [21].

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Aminabadi, R.Y., Rajbhandari, S., Awan, A.A., et al.: Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale. In: SC (2022). https://doi.org/10.1109/SC41404.2022.00051
2. Anil, R., Borgeaud, S., Wu, Y., et al.: Gemini: A family of highly capable multimodal models. arXiv (2023). https://doi.org/10.48550/ARXIV.2312.11805
3. Bavarian, M., Jun, H., Tezak, N., et al.: Efficient training of language models to fill in the middle. arXiv (2022). https://doi.org/10.48550/ARXIV.2207.14255
4. Brown, T.B., Mann, B., Ryder, N., et al.: Language models are few-shot learners. In: NeurIPS (2020)
5. OpenAI canvas. https://openai.com/index/introducing-canvas/
6. Chen, M., Tworek, J., Jun, H., et al.: Evaluating large language models trained on code. arXiv (2021)
7. Chowdhery, A., Narang, S., Devlin, J., et al.: Palm: Scaling language modeling with pathways. J. Mach. Learn. Res. (2023)
8. Amazon CodeWhisper. https://docs.aws.amazon.com/codewhisperer/
9. GitHub Copilot. https://github.com/features/copilot
10. Dao, T.: Flashattention-2: Faster attention with better parallelism and work partitioning. In: ICLR (2024)
11. Dao, T., Fu, D.Y., Ermon, S., et al.: Flashattention: Fast and memory-efficient exact attention with io-awareness. In: NeurIPS (2022)
12. DeepSeek-AI, Liu, A., Feng, B., Xue, B., et al.: Deepseek-v3 technical report. arXiv (2024). https://doi.org/10.48550/ARXIV.2412.19437
13. DeepSeek V3 serving. https://zhuanlan.zhihu.com/p/27181462601
14. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. In: NAACL-HLT (2019). https://doi.org/10.18653/V1/N19-1423
15. Ding, Y., Wang, Z., Ahmad, W.U., et al.: Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In: NeurIPS (2023)
16. Dubey, A., Jauhri, A., Pandey, A., et al.: The llama 3 herd of models. arXiv (2024). https://doi.org/10.48550/ARXIV.2407.21783
17. Fried, D., Aghajanyan, A., Lin, J., et al.: Incoder: A generative model for code infilling and synthesis. In: ICLR (2023)
18. Gao, B., He, Z., Sharma, P., et al.: Cost-efficient large language model serving for multi-turn conversations with cachedattention. In: ATC (2024)

19. Gim, I., Chen, G., Lee, S., et al.: Prompt cache: Modular attention reuse for low-latency inference. In: MLSys (2024)
20. Guo, D., Zhu, Q., Yang, D., et al.: Deepseek-coder: When the large language model meets programming - the rise of code intelligence. arXiv (2024). https://doi.org/10.48550/ARXIV.2401.14196
21. Guo, T., Dong, H., Leng, Y., Liu, F., Lin, C., Xiao, N., Zhang, X.: EFIM: Efficient Serving of LLMs for Infilling Tasks with Improved KV Cache Reuse (Jun 2025). https://doi.org/10.5281/zenodo.15580572
22. Hui, B., Yang, J., Cui, Z., et al.: Qwen2.5-coder technical report. arXiv (2024). https://doi.org/10.48550/ARXIV.2409.12186
23. Kwon, W., Li, Z., Zhuang, S., et al.: Efficient memory management for large language model serving with pagedattention. In: SOSP (2023). https://doi.org/10.1145/3600006.3613165
24. Lee, W., Lee, J., Seo, J., Sim, J.: Infinigen: Efficient generative inference of large language models with dynamic KV cache management. In: OSDI (2024)
25. Li, R., Allal, L.B., Zi, Y., et al.: Starcoder: may the source be with you! TMLR (2023)
26. Llama3.1 model card. https://huggingface.co/meta-llama/Llama-3.1-8B
27. Lozhkov, A., Li, R., Allal, L.B., et al.: Starcoder 2 and the stack v2: The next generation. arXiv (2024). https://doi.org/10.48550/ARXIV.2402.19173
28. Nijkamp, E., Hayashi, H., Xiong, C., et al.: Codegen2: Lessons for training llms on programming and natural languages. arXiv (2023). https://doi.org/10.48550/ARXIV.2305.02309
29. Nijkamp, E., Pang, B., Hayashi, H., et al.: Codegen: An open large language model for code with multi-turn program synthesis. In: ICLR (2023). https://doi.org/10.48550/ARXIV.2312.11805
30. Pope, R., Douglas, S., Chowdhery, A., et al.: Efficiently scaling transformer inference. In: MLSys (2023)
31. Rozière, B., Gehring, J., Gloeckle, F., et al.: Code llama: Open foundation models for code. arXiv (2023). https://doi.org/10.48550/ARXIV.2308.12950
32. Starcoderdata. https://huggingface.co/datasets/bigcode/starcoderdata
33. Vaswani, A., Shazeer, N., Parmar, N., et al.: Attention is all you need. In: NeurIPS (2017)
34. Wang, X., Wang, Z., Liu, J., et al.: MINT: evaluating llms in multi-turn interaction with tools and language feedback. In: ICLR (2024)
35. Wang, Y., Chen, K., Tan, H., Guo, K.: Tabi: An efficient multi-level inference system for large language models. In: EuroSys (2023). https://doi.org/10.1145/3552326.3587438
36. Xiao, G., Tian, Y., Chen, B., et al.: Efficient streaming language models with attention sinks. In: ICLR (2024)
37. Ye, L., Tao, Z., Huang, Y., Li, Y.: Chunkattention: Efficient self-attention with prefix-aware KV cache and two-phase partition. In: ACL (2024). https://doi.org/10.18653/V1/2024.ACL-LONG.623
38. Yu, G., Jeong, J.S., Kim, G., et al.: Orca: A distributed serving system for transformer-based generative models. In: OSDI (2022)
39. Yu, L., Lin, J., Li, J.: Stateful large language model serving with pensieve. In: EuroSys (2025). https://doi.org/10.1145/3689031.3696086
40. Zhang, Z., Sheng, Y., Zhou, T., et al.: H2O: heavy-hitter oracle for efficient generative inference of large language models. In: NeurIPS (2023)
41. Zheng, L., Yin, L., Xie, Z., et al.: Sglang: Efficient execution of structured language model programs. In: NeurIPS (2024)