FastCoder: Accelerating Repository-level Code Generation via Efficient Retrieval and Verification

Qianhui Zhao[1], Li Zhang[1], Fang Liu[1*], Xiaoli Lian[1*], Qiaoyuanhe Meng[1]
Ziqian Jiao[1], Zetong Zhou[1], Jia Li[3], Lin Shi[2]
[1]State Key Laboratory of Complex & Critical Software Environment, School of Computer Science and Engineering
Beihang University, China
[2]School of Software, Beihang University, China
[3]Peking University, China
Email: {zhaoqianhui, fangliu, lianxiaoli}@buaa.edu.cn

***Abstract*—Code generation is a latency-sensitive task that demands high timeliness. However, with the growing interest and inherent difficulty in repository-level code generation, most existing code generation studies focus on improving the correctness of generated code while overlooking the inference efficiency, which is substantially affected by the overhead during LLM generation. Although there has been work on accelerating LLM inference, these approaches are not tailored to the specific characteristics of code generation; instead, they treat code the same as natural language sequences and ignore its unique syntax and semantic characteristics, which are also crucial for improving efficiency. Consequently, these approaches exhibit limited effectiveness in code generation tasks, particularly for repository-level scenarios with considerable complexity and difficulty. To alleviate this issue, following draft-verification paradigm, we propose FastCoder, a simple yet highly efficient inference acceleration approach specifically designed for code generation, without compromising the quality of the output. FastCoder constructs a multi-source datastore, providing access to both general and project-specific knowledge, facilitating the retrieval of high-quality draft sequences. Moreover, FastCoder reduces the retrieval cost by controlling retrieval timing, and enhances efficiency through parallel retrieval and a context- and LLM preference-aware cache. Experimental results show that FastCoder can reach up to $2.53\times$ and $2.54\times$ speedup compared to autoregressive decoding in repository-level and standalone code generation tasks, respectively, outperforming state-of-the-art inference acceleration approaches by up to 88%. FastCoder can also be integrated with existing correctness-focused code generation approaches to accelerate the LLM generation process, and reach a speedup exceeding $2.6\times$.**

***Index Terms*—code generation, inference acceleration, large language models, retrieval-augmented generation**

I. Introduction

Code generation is a crucial task in software engineering, which can reduce manual effort by automating repetitive and time-consuming programming tasks. Large Language Models (LLMs), such as DeepSeek-Coder [1], CodeLlama [2], GPT-4o [3], *etc.*, have demonstrated impressive performance in code generation tasks, revolutionizing the landscape of software development [4, 5]. Substantial researches have been proposed to improve the correctness of code (evaluated by the passing rate [6]) generated by LLMs [7, 8, 9]. While code correctness ensures the program performs its intended behaviors accurately, efficiency of code generation is also crucial for real-world software development but often overlooked. Therefore, optimizing the efficiency of code generation approaches has become a pressing issue.

In real-world software development, developers typically work within a specific context—such as a repository—that contains rich domain knowledge. Since target code often relies on functions or classes already defined in the repository, effective repository-level code generation must consider the context of the entire repository. However, due to the large scale of code repositories and the limited context window of LLMs, it is infeasible to directly leverage the entire repository as input. Thus, most existing studies adopt a retrieval-augmented generation (RAG) strategy [9, 10, 11], using the natural language requirement or partially written code as a query to retrieve relevant code snippets from the repository, which are then combined with the original query to guide code generation.

The RAG strategy can bring benefits to the functional correctness of the generated code, but neglects generation efficiency. The latency of this paradigm is primarily composed of two stages: the retrieval stage and generation stage. As we all know, during the generation stage, LLMs face the challenge of the significant inference time caused by autoregressive decoding mechanism. In this process, each new token is generated sequentially, conditioned on all previously generated tokens and the given context, resulting in repeated and computationally expensive forward passes. Moreover, the growing complexity of relevant context retrieval and prompt construction in RAG approaches further increases the overall time cost of code generation. However, developers typically hold high expectations regarding the responsiveness of code recommendations [12]. If LLMs fail to deliver precise and efficient feedback, it may directly affect development efficiency and user experience.

Although there exist approaches for LLM inference acceleration, they exhibit limited effectiveness in code generation, especially in repository-level code generation. For example, Self-speculative decoding [13] achieves approximately $1.5\times$ acceleration compared to autoregressive decoding in stan-

* Corresponding author

dalone code generation (Fig. 5), but falls short when applied to repository-level tasks, offering virtually no speedup (Table I). Most of these approaches follow a draft-verification paradigm [13, 14, 15, 16], which utilizes a small model or a retrieval system to draft several candidate output tokens, and then employs the target LLM to verify the acceptability of draft tokens through a single forward step, keeping the output consistent with that decoded autoregressively by the target LLM itself. The suboptimal performance on repository-level code generation may be attributed to the reason that existing approaches treat source code as sequences similar to natural language, without accounting for code's unique syntactic and semantic characteristics. For instance, the fixed retrieval datastore of retrieval-based acceleration approach struggles to provide high-quality drafts for different repositories that follow customized functions and coding styles, and cannot adapt to output preferences of different LLMs. Additionally, performing retrieval uniformly across all code locations overlooks the varying difficulty levels of code generation at different positions. Therefore, **there lack acceleration techniques specifically tailored for repository-level code generation tasks, which are orthogonal to correctness-focused approaches, enabling improvements in inference speed without compromising correctness performance.**

To alleviate this issue, in this paper, we primarily focus on improving the inference speed of LLMs on repository-level code generation task, without compromising the quality of the output. We propose FASTCODER, a **simple yet highly efficient** approach to accelerate the inference of LLMs through an efficient and effective retrieval strategy. Concretely, to better align with the characteristics of different repositories, we first construct a multi-source datastore, providing access to both general and project-specific knowledge and enhancing the quality of draft sequences. Then, FASTCODER reduces unnecessary retrieval overhead by controlling the retrieval timing. Besides, FASTCODER improves retrieval efficiency through parallel retrieval and the maintenance of a context- and LLM preference-aware cache. Finally, draft sequences are constructed using a weighted Trie, and tree attention is employed to avoid redundant computation caused by verifying multiple draft sequences.

Experimental results show that the decoding speed of FASTCODER surpasses existing inference acceleration approaches substantially on both repository-level and standalone code generation tasks. For repository-level code generation, FASTCODER achieves up to $2.30\times$ and $2.53\times$ speedup compared with autoregressive decoding on DevEval [17] and RepoEval [11], respectively. FASTCODER can also achieve up to $2.54\times$ acceleration on standalone code generation dataset, HumanEval [6]. Moreover, it is worth mentioning that FASTCODER can be integrated with existing functional correctness-focused code generation methods to improve their generation efficiency while preserving the consistency of output sequences. When combined with RepoCoder [11] and RLCoder [18], FASTCODER can bring a speedup exceeding $2.6\times$.

Our contributions can be summarized as follows:

- We identify limitations of current correctness-focused code generation studies and LLM inference acceleration approaches in repository-level code generation and provide insights for potential improvements.
- We propose FASTCODER, a simple yet efficient approach to accelerate LLM inference for code generation by leveraging effective retrieval and verification mechanisms, which can also be integrated with correctness-focused code generation approaches following RAG strategy.
- We conduct a comprehensive evaluation of the inference efficiency of FASTCODER, and results show that it achieves state-of-the-art results in both repository-level and standalone code generation tasks. We provide the code, data, and an initial demo at https://github.com/whisperzqh/FastCoder.

II. RELATED WORK

A. Repository-level Code Generation

LLMs have made significant progress in recent years, yet widely-used evaluation datasets for code generation task, such as HumanEval [6] and MBPP [19], predominantly focus on standalone code generation tasks. This over-simplified setting falls short of representing the real-world software development scenario where repositories span multiple files with numerous cross-file dependencies. In practice, a substantial portion of code relies on methods and properties defined in other files. These non-standalone functions constitute more than 70% of the functions in popular open-source projects, making evaluations based solely on standalone functions insufficient for assessing model effectiveness [20].

Consequently, repository-level code generation has gained increasing attention due to the inter-dependencies among code files, such as cross-module API calls and shared global snippets, which are crucial for context-aware code generation. However, due to the vast size of code repositories and the limitations of LLMs' context length, it is impossible to leverage the entire repository directly as context. Most prior studies adopt a retrieval-augmented generation (RAG) strategy, using the natural language requirement or unfinished code as a query to retrieve snippets from the repository, which are then combined with the query for code generation. RepoFuse [7] employs BM25 [21] for lexical retrieval based on textual similarity with the unfinished code. RepoHyper [8] adopts dense retrieval by encoding code into vectors to capture semantic similarity. RepoCoder [11] enhances retrieval iteratively using intermediate completions to better align context and generation. RLCoder [18] leverages reinforcement learning to train the retriever without labeled data. As context retrieved based on text similarity may be insufficiently relevant to generation targets, Draco [10] constructs a repo-specific context graph via dataflow analysis to precisely retrieve relevant background knowledge. GraphCoder [9] integrates graph-based retrieval with LLMs to combine general and repository-specific knowledge for code generation. However, these approaches focus solely on improving the functional correctness of the generated code, while neglecting generation efficiency. The construction

of information-rich prompts further increases the overall time cost of code generation.

Researchers have also introduced several benchmarks to evaluate code generation in realistic scenarios. RepoEval [11] focuses on repository-level code completion covering line, API invocation, and function body granularities. CoderEval [20] is a context-aware benchmark that categorizes code generation tasks into six levels based on external dependencies, ranging from standalone to project-level functions. CrossCodeEval [22] is a multilingual benchmark that emphasizes cross-file contextual understanding and employs static analysis to identify examples requiring such context. DevEval [17] aligns with real-world repositories in terms of code and dependency distributions, and includes comprehensive metadata annotated manually by developers.

B. LLM inference acceleration approaches

Autoregressive decoding generates tokens in a step-by-step manner and results in a slow and costly decoding process. To accelerate decoding, previous non-autoregressive decoding approaches [12, 23] were proposed to generate multiple tokens in parallel. However, this parallelism often comes at the cost of degraded model performance. To address this issue, researchers have explored decoding acceleration methods that preserve model performance. Among these, draft-verification approaches [15, 16, 24] have gained widespread adoption, which employ an alternative method to quickly produce candidate output tokens and then utilize the target LLM to verify them in a single forward pass. These methods can be further categorized into generation-based and retrieval-based approaches, depending on how the initial draft is produced.

For generation-based approaches, the draft token can be generated either by a small model or the target LLM itself. Speculative decoding [24, 25] minimizes the target LLM forward steps by using a smaller model for drafting and then employing the target LLM to verify the draft in a low-cost parallel manner. Based on this, Ouroboros [14] generates draft phrases to parallelize the drafting process and lengthen drafts. Specinfer [15] uses many draft models obtained from distillation, quantization, and pruning to conduct speculations together. However, identifying an appropriate draft model continues to pose significant challenges, as it must align with the vocabulary of the target LLM and achieve a delicate balance between keeping quick decoding speed and ensuring output quality. Thus, researchers have investigated utilizing the target LLM itself to generate efficient draft sequences. Blockwise Decoding [26] and Medusa [27] introduce multiple heads to enable parallel generation of multiple tokens per step. Lookahead decoding [28] uses an n-gram pool to cache the historical n-grams generated so far. Eagle [29] conducts the drafting process at the more structured feature level. Self-speculative decoding [13] employs the target LLM with selectively certain intermediate layers skipped as the draft model. However, using the target LLM itself to generate draft tokens often requires modifications to the model architecture, necessitating additional training to achieve optimal performance.

To avoid additional training and identifying a suitable draft model, as well as the extra computational overhead introduced by draft models, more recently, researchers have explored retrieval-based approaches, which replace the draft model with a retrieval system [16, 30]. By searching in a retrieval datastore to obtain candidate sequences, these approaches can easily be ported to any LLM and reduce computational overhead. LLMA [30] is an inference-with-reference decoding mechanism by exploiting the overlap between the output and the reference of an LLM. It provides generic speedup through speculative retrieval and batched verification. REST [16] replaces the parametric draft model with a non-parametric retrieval datastore. As many subsequences during generation likely appear in the datastore, it can frequently generate multiple correct tokens per step.

Our approach adopts the retrieval-based acceleration paradigm following REST [16], but has distinctive innovations, as it is specifically tailored to the characteristics of code generation. In particular, we design a multi-source datastore to reduce retrieval overhead while enhancing the relevance of retrieved results, exploit structural properties of code to avoid unnecessary retrieval in positions where it is unlikely to be beneficial, and incorporate a caching mechanism to ensure that retrieved drafts are better aligned with the preferences of the LLM and the characteristics of the target repository.

III. Preliminaries

A. Retrieval-based Speculative Decoding

Building upon the draft-verification framework introduced by speculative decoding [24, 25], retrieval-based decoding acceleration approaches leverage a retrieval mechanism to generate draft tokens [16, 30], which can eliminate the challenge of selecting an appropriate draft model and avoid additional training costs. A notable example is Retrieval-Based Speculative Decoding (REST) [16], which has proven to be effective in standalone function generation task [6]. Below is an explanation of how it works. Pre-built from a code corpus, the datastore $D = \{(c_i, t_i)\}$ serves as the source for the draft token sequence, where c_i represents a context and t_i represents the corresponding continuation of c_i. As an alternative to the draft model, the objective of retrieval is to identify the most likely continuations of the current context from the datastore D using a suffix match [31]. Specifically, given a context $s = (x_1, ..., x_t)$, it aims to find contexts in D that match the longest suffix of s. Starting from a pre-defined match length upper limit n_{max} (measured in the number of tokens), for each suffix length n, it extracts the suffix of s with n tokens, denoted as q, and obtains all contexts c_i that match q as a suffix. If at least one context in D matches q, the corresponding context continuation pairs are returned as the retrieval result S; otherwise, the match length n is decreased by one to attempt matching a shorter suffix. Subsequently, the top-k high-frequency prefixes in S are selected as the draft sequences for later verification. Inspired by REST, FastCoder also incorporates a similar suffix-match-based retrieval algorithm, leveraging its advantages in time and memory efficiency.

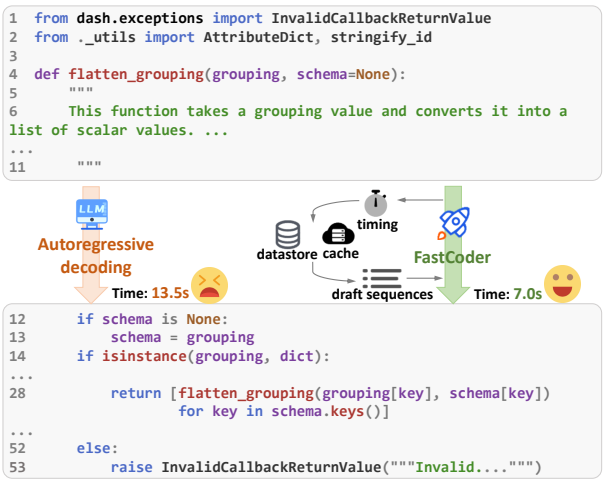


Fig. 1: Time overhead of a practical code generation example using CodeLlama-7B on a single NVIDIA 4090 GPU.

```
def _get_user_id():
    """
    Get the user id from the user id file. If the user id file does
not exist or is older than 24 hours, generate a new user id and save
it to the user id file.
    Input-Output Arguments
    :return: String. The user id.
    """
    user_id_file_path = os.path.join(config.get_config_dir(),
TELEMETRY_ID_FILE)
    if _user_id_file_is_old(user_id_file_path):
        user_id = str(uuid.uuid4())
        with open(user_id_file_path, 'w') as f:
            f.write(user_id)
    else:
        with open(user_id_file_path, 'r') as f:
            user_id = f.read()
    return user_id
```

- fail to retrieve *user_id_file_path*
- can only retrieve *file_path*

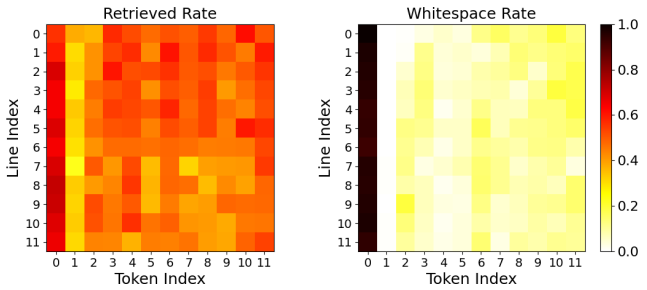
Fig. 2: Localness of source code.

B. Motivating Examples

We first present a practical example in Figure 1. Given a function signature and its context as input, autoregressive decoding with an LLM (CodeLlama-7B) takes about 13.5 seconds on a single NVIDIA 4090 GPU to generate the function body, where such latency may negatively affect user experience. This highlights the need for efficient methods to accelerate code generation. To identify the limitations of current inference acceleration methods, we present motivating examples that highlight the localness of source code and the retrieval performance in retrieval-based approaches.

Localness of source code. Human-written programs are typically localized [32], with program entities (token sequences) defined or used in the preceding snippets frequently being reused in the subsequent code snippets within the same code file. As shown in Fig. 2, *user_id_file_path* is a user-defined variable within the current code segment, which does not exist in the datastore but appears multiple times in subsequent code snippets. Additionally, the blue-highlighted statements demonstrate the repetition of token sequences. *By effectively leveraging these frequently occurring token sequences within the file, such as storing them in a cache for subsequent retrieval, the acceptance length for draft validation can be increased, thereby enhancing the inference speed.*

Retrieval is not always essential. Current work performs retrieval operation at *every* position, which may bring unnecessary cost. To investigate the relationship between retrieval performance and token position in code generation, we randomly selected 200 samples from DevEval [17], a repository-level code generation benchmark, and employed DeepSeek-Coder-6.7B [1] for evaluation. For each token, we recorded whether it was: (a) retrieved from the datastore rather than generated by the model, and (b) a whitespace character (e.g., spaces or newline characters). Results are presented as heatmaps in Fig. 3. As seen from Fig. 3(a), *retrieval failures are frequent, with a particularly notable pattern: the second token in each line has the lowest probability of being successfully retrieved*. A comparison with the whitespace rate heatmap suggests that this phenomenon may stem from the fact that the second token is typically the first non-whitespace character at the beginning of a line. The first non-whitespace token in each line dictates the direction of the line, making it more variable and consequently more challenging to retrieve. Thus, *skipping retrieval or reducing the retrieval probability at such positions may improve performance*.



(a) (b)

Fig. 3: Heatmaps of (a) retrieval performance and (b) whitespace distribution with token positions in REST. The maximum token index is selected based on the average token number per line (12).

Motivated by the above observations, we propose FastCoder, which is specifically tailored for code generation. As shown in Figure 1, FastCoder significantly reduces the function body generation time from 13.5s to 7.0s.

IV. Approach

To improve the inference efficiency of LLMs on code generation tasks, we propose FastCoder, an LLM inference acceleration approach tailored to the characteristics of code generation. The architecture of FastCoder is shown in Fig. 4. FastCoder constructs a multi-source datastore combining general and repository-specific knowledge to improve draft quality. It then reduces retrieval overhead by controlling timing, boosts efficiency via parallel retrieval and a context and LLM preference-aware cache. Finally, it uses tree attention to avoid redundant computation.

A. Multi-source Datastore Construction

The quality of the retrieval datastore, which serves as the source of draft sequences, critically determines the acceleration potential. A larger datastore may enhance the probability

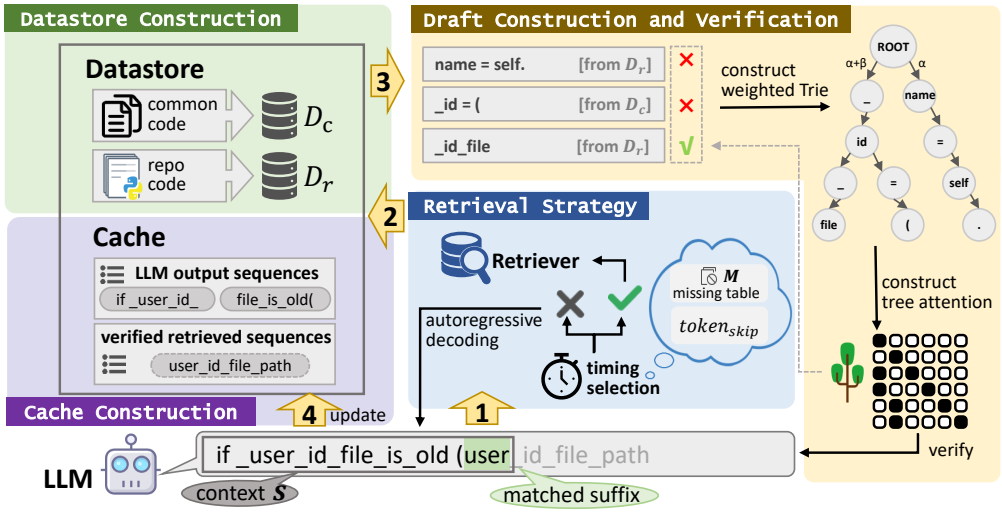


Fig. 4: Architecture of FASTCODER.

of result acceptance, but it also correspondingly increases retrieval time, making the trade-off between the two critically important. To achieve optimal performance with a compact datastore and facilitate effective retrieval, FASTCODER incorporates a smaller repository-related datastore D_r and a larger common code datastore D_c to construct a comprehensive retrieval datastore D. This design supports parallel retrieval, providing access to both general and project-specific knowledge. To enable fast retrieval with minimal overhead, we organize the datastore into context-continuation pairs, facilitating a rapid exact-match method for context search.

Repository-related datastore D_r. During software development, developers often reference cross-file elements such as classes and methods, making intra-repository files highly relevant to the generated code. Additionally, repository-specific factors, including domain variations and coding conventions, lead to distinct patterns of idiomatic expressions. For instance, web development repositories frequently involve HTTP request-response handling, while data science repositories focus on data processing and modeling tasks. To this end, we collect the code files from current repository (with the portions to be generated excluded to avoid data leakage) and form repository-related datastore D_r.

Common datastore D_c. To ensure that common programming operations are also retrievable, a subset of data from commonly used pre-trained code datasets [33] is used to form D_c, which serves as another component of datastore D.

Datastore organization. For efficient retrieval, the datastore is organized as contexts and the corresponding continuations following [16]. Specifically, for each code file utilized in constructing the datastore, the content preceding every position will constitute a context, whereas the content subsequent to that position is the corresponding continuation. The datastore D of FASTCODER can be summarized as:

$$D = (D_r, D_c) \tag{1}$$

$$(D_r, D_c) = (\{(c_i, t_i)\}_{i=1}^{|D_r|}, \{(c_j, t_j)\}_{j=1}^{|D_c|}) \tag{2}$$

where c_i (c_j) represents the context, t_i (t_j) represents the corresponding continuation of c_i (c_j), $|D_r|$ ($|D_c|$) is the number of samples in D_r (D_c). Specifically, D_r can be omitted in standalone code generation where such context is unreachable.

B. Context- and LLM Preference-aware Caching

To reduce retrieval costs and improve the alignment of retrieved results with context and LLM preferences—thereby increasing both the accepted sequence length and inference speed—we design a context- and LLM preference-aware caching strategy to cache the verified retrieved sequences and LLM-generated sequences.

Based on the observations in Section III-B that program entities (token sequences) defined or used in preceding snippets are often reused in the subsequent code snippets, we design the CACHE mechanism from two perspectives. On the one hand, although datastore D contains vast content, typically only a small portion is highly relevant to the code currently being generated. In contrast, the validated retrieval tokens are exactly the sequences that exhibit high relevance to the current generation code, and are thus more likely to be retrieved again in subsequent stages than other content within the datastore D. Consequently, if the draft sequence $r = (y_1, ..., y_j)$, retrieved by the context $s = (x_1, ..., x_t)$, is verified by the LLM, we concatenate them as $(x_1, ..., x_t, y_1, ..., y_j)$ and add it into CACHE. On the other hand, since the datastore D is static, as it remains unmodified after construction, the draft sequences retrieved for the identical context s also remain consistent. However, different LLMs exhibit distinct generation preferences, which is reflected in the fact that they may generate different outputs given the same input. As a result, a static datastore struggles to accommodate the diverse preferences of various LLMs. Earlier decoding outputs can to some extent reflect LLM-specific tendencies and ensure contextual coherence. Therefore, we also incorporate the verified decoding output sequence into CACHE for future use.

To maintain the CACHE, we assess whether the two aforementioned update conditions are satisfied after each forward step of the LLM. If the number of sequences inside the CACHE

exceeds the pre-defined threshold l, it is accessible and will remain active throughout the entire inference process.

C. Dynamic and Efficient Retrieval Strategy

Algorithm 1 illustrates the complete retrieval process of FASTCODER. Before each forward step, given current context s, FASTCODER initially verifies the availability of CACHE. If the CACHE is accessible, that is, the number of sequences inside exceeds l, retrieval is prioritized from CACHE. If CACHE is unavailable or fails to yield valid (non-empty) results, FASTCODER utilizes a dynamic and efficient retrieval strategy to minimize unnecessary retrieval cost. Specifically, FASTCODER optimizes retrieval timing by addressing two key considerations as follows.

Skip token. As mentioned in Section III-B, the intrinsic characteristics of code lead to a low retrieval success rate at the first non-whitespace character of each line. Since obvious patterns are not found in other positions, and the introduction of intricate judgment processes may incur additional computational overhead, we set the first non-whitespace character of each line as the skip token. We strategically reduce the retrieval probability of skip tokens through a control parameter p, which refers to the retrieval probability at these positions.

Missing table. When utilizing the current context s to retrieve its continuations from datastore D, it may fail to yield any valid results in some cases. To prevent time wastage resulting from invalid retrieval, we maintain a missing table $M = \{s_{m_i}\}$ that stores suffixes s_{m_i} for which no valid results can be retrieved from the datastore D. Thus, when s_{m_i} is encountered again during the subsequent inference, FASTCODER will bypass the retrieval and directly utilize the LLM to generate the next token.

If FASTCODER decides to proceed with retrieval according to the above strategy, parallel retrieval is conducted from D_r and D_c to further boost the retrieval efficiency, and the results refer to R_r and R_c, separately. Specifically, if R_r and R_c are both empty, s will be denoted as s_m and added into the missing table M. Otherwise, relevant sequences are employed to update the CACHE.

D. Draft Construction and Verification with Weighted Prefix Optimization

The retrieval results $R = (R_r, R_c)$ contain potential continuations of the current context s, often sharing the same prefix. To reduce the cost brought by verification each $r_i \in R$ one by one, we construct the draft sequences using a Trie, where the unique path from a node to the root node corresponds to a prefix of the retrieval results, aiming to reduce the repeated verification of shared prefixes in R. We use following equation to assign a weight for each node:

$$N_{weight} = \alpha \cdot t_r + \beta \cdot t_c \tag{3}$$

where t_r and t_c represents the times that the node occurs in R_r and R_c respectively, and α and β refers to the corresponding coefficient. We retain α and β as tunable parameters to accommodate diverse scenarios. For instance, in highly specialized code generation tasks, increasing the α/β value can help generate draft sequences that better align with repository-specific characteristics. We select top-k weighted sequences from the Trie as the draft sequences.

Algorithm 1: Retrieval Algorithm

Input: current context s, datastore D, retrieval cache CACHE, minimum activation size l, missing table M, skip token $token_{\text{skip}}$, retrieval probability p
Output: Retrieved sequences R

```
if CACHE.size ≥ l then
    // retrieval from cache
    R ← search(CACHE)
if CACHE.size < l or R = ∅ then
    // retrieval timing selection
    if s ∈ M then
        pass
    else if s ends with token_skip then
        if random number < p then
            // parallel retrieval from datastore
            R_r, R_c ← par_search(D_r, D_c)
            R ← (R_r, R_c)
if R = ∅ then
    // update missing table
    M ← M ∪ {s}
else
    update CACHE
return R;
```

As many draft sequences may share common prefixes, to avoid redundant computation of Transformer layers, we employ tree attention [15, 34] to verify the draft sequences. We represent the tree formed by the draft sequences, which is part of the Trie, as $\mathcal{T} = (V, E)$, where V is the set of all generated tokens and E is the set of edges representing token transitions in the candidates. The total number of nodes is $N = |V|$. In tree attention, only a token's predecessors are considered as historical context, and the attention mask restricts attention to these predecessors. Let $\mathbf{A} \in \{0,1\}^{N \times N}$ be the attention mask matrix, where $A_{ij} = 1$ indicates that token i can attend to token j. To reflect the tree structure, we define:

$$A_{ij} = \begin{cases} 1, & \text{if token } j \text{ is a predecessor of token } i \text{ in } \mathcal{T} \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

This ensures that each token can only attend to its own continuation path, and tokens from different candidate branches are isolated in the attention computation.

By applying tree attention mask and appropriately adjusting the positional indices for encoding, we are able to process multiple candidates simultaneously without increasing the batch size. As our objective is to accelerate the inference without compromising model performance, all correct tokens from the beginning will be accepted, while the draft tokens following the first error will be rejected.

V. EVALUATION

To evaluate the effectiveness of FASTCODER, we address the following research questions:

RQ1: Overall Performance. How does the inference acceleration performance of FASTCODER compare against state-of-the-art approaches?
RQ2: Ablation Study. What are the contributions of each component of FASTCODER?
RQ3: Integration with Correctness-Focused Approaches. Can FASTCODER enhance generation speed when integrated with existing code generation methods that focus on the code correctness improvement?

A. Datasets and Backbone Models.

FASTCODER is not only applicable to repository-level code generation, but can also be adapted to standalone code generation with minimal modifications. Thus, we conduct experiments on both repository-level and standalone code generation benchmarks. For repository-level code generation, we choose two widely-used benchmarks, DevEval [17] and RepoEval [11]. DevEval comprises 1,825 testing samples from 115 repositories, covering 10 popular domains. It aligns with real-world repositories in code distributions and dependency distributions. RepoEval is constructed using the high-quality repositories sourced from GitHub. We use the function-level subset for evaluation, which contains 455 testing samples. For standalone code generation, we conduct experiments on HumanEval [6], a widely-used standalone code generation dataset including 164 human-written programming problems.

For backbone models, we use the 1.3B and 6.7B configurations of Deepseek-Coder-base [1], as well as 7B and 13B configurations of CodeLlama-Python [2] for evaluation, which are popular and well-performing LLMs in code generation.

B. Baselines.

We compare FASTCODER with vanilla **Autoregressive decoding** and the following state-of-the-art inference acceleration approaches that follow the draft-verification framework and have demonstrated effectiveness in code generation:

- **Self-speculative decoding** [13]: This approach is generation-based and employs the target LLM with selectively certain intermediate layers skipped as the draft model, without the need for an auxiliary model.
- **Ouroboros** [14]: This is a generation-based approach and demands manual selection of a suitable draft model for the target LLM. To maximize the utility of discarded drafts, it generates phrases from them to parallelize the drafting process and lengthen drafts in a training-free manner.
- **REST** [16]: REST is a retrieval-based and draft model-free approach, which draws from the reservoir of existing knowledge, retrieving and employing relevant tokens based on the current context.

C. Evaluation Metrics.

Following existing work [13, 14, 16], we adopt the following metrics to assess the inference efficiency:

- **Decoding speed (ms/token)**: The average generation time of one token for the LLM with batch size set to 1 on a single GPU. Formally, if L denotes the length of the generated tokens and T represents the time spent for generation, decoding speed is computed by:

$$\textit{Decoding Speed} = T/L \tag{5}$$

- **Speedup**: The ratio by which the decoding speed of an evaluated method is increased compared to vanilla autoregressive decoding. Specifically, if $Speed_A$ and $Speed_B$ represent the decoding speed of vanilla autoregressive decoding and the evaluated approach, respectively, the speedup can be calculated by:

$$\textit{Speedup} = Speed_A/Speed_B \tag{6}$$

- **Average Acceptance Length**: The average number of tokens accepted per forward step by the target LLM, which represents the theoretical upper bound of achievable acceleration for retrieval-based approaches. Formally, if L denotes the length of generated tokens, and F represents the number of forward steps by the target LLM, it can be calculated by:

$$\textit{Average Acceptance Length} = L/F \tag{7}$$

Although FASTCODER can generate code theoretically consistent with the target LLM's output, we still evaluate the correctness of the generated code to enable quantitative assessment. For DevEval [17] and HumanEval [6], we report **Pass@1** results. For RepoEval [11], we report **Edit Similarity (ES)** results, since Pass@k scripts are unavailable due to licensing as stated in their GitHub repository.

D. Implementation Details.

To provide essential contextual information, we prepend preceding code snippets from the same file as context for DevEval and RepoEval. Following existing works [11, 16, 17], all results are obtained with a maximum input length of 2k and generation length of 512 under greedy decoding. D_c is constructed from the first Python file of pre-training code in The Stack [33], taking approximately 9 minutes and yielding a 0.9GB datastore. D_r ranges from 60KB to 289MB across repositories, taking an average of 10 seconds. Based on preliminary experiments, we set the hyper-parameters to $l = 50$, $p = 0.5$, and $\alpha = \beta = 1$, with LLM output truncated every 20 tokens and added to the CACHE. Evaluation of p over the range 0.1 to 0.9 and l from 10 to 100 showed an initial improvement followed by a decline, and the final values were chosen for best performance. α and β were similarly selected from a range of tested combinations. Following [16], for retrieval, the starting context suffix length $n_{max} = 16$, and a maximum of 64 draft tokens of the top-k sequences are selected in the Trie. All experiments with Deepseek-Coder-1.3B/6.7B and CodeLlama-7B use a single NVIDIA 4090 GPU and 28 CPU cores, and CodeLlama-13B experiments use a single NVIDIA A6000 GPU and 12 CPU cores.

The implementation details of baseline approaches can be seen as follows. For Self-speculative decoding, we adopt the skipped layers from [13] for CodeLlama-13B. For DeepSeek-Coder and CodeLlama-7B, where specific layers are not provided, we follow the instruction to determine them, which takes several hours. For Ouroboros, while selecting the draft

TABLE I: Decoding speed and speedup ratio on repository-level code generation datasets. For FASTCODER, we additionally report the *average per-sample decoding time* and *Pass@1 / ES score*, with the comparison to autoregressive decoding shown in the upper-right corner, which are highlighted with a blue background ($Avg.Time/Pass@1^{\Delta v.s.AR}$ for DevEval and $Avg.Time/ES^{\Delta v.s.AR}$ for RepoEval).

Dataset	Approach	Deepseek-Coder-1.3B ms/token	Deepseek-Coder-1.3B Speedup	Deepseek-Coder-6.7B ms/token	Deepseek-Coder-6.7B Speedup	CodeLlama-7B ms/token	CodeLlama-7B Speedup	CodeLlama-13B ms/token	CodeLlama-13B Speedup
DevEval	Autoregressive	20.00	1.00×	26.15	1.00×	26.29	1.00×	46.35	1.00×
	Self-speculative	18.72	1.07×	22.55	1.16×	25.10	1.05×	42.74	1.08×
	Ouroboros	-	-	15.69	1.67×	29.14	0.90×	39.73	1.17×
	REST	12.10	1.65×	15.28	1.71×	15.57	1.69×	43.38	1.07×
	FASTCODER	**8.71**	**2.30**×	**11.69**	**2.24**×	**12.17**	**2.16**×	**21.56**	**2.15**×
		$4.4s^{\downarrow 5.8s}$ / 20.38^{-}		$6.0s^{\downarrow 7.4s}$ / 27.01^{-}		$6.2s^{\downarrow 7.3s}$ / 29.81^{-}		$11.0s^{\downarrow 12.7s}$ / 30.90^{-}	
RepoEval	Autoregressive	19.91	1.00×	25.75	1.00×	26.21	1.00×	47.86	1.00×
	Self-speculative	19.63	1.02×	22.48	1.16×	24.36	1.08×	42.09	1.14×
	Ouroboros	-	-	14.56	1.77×	33.12	0.79×	35.60	1.34×
	REST	12.09	1.65×	15.46	1.67×	15.43	1.70×	44.59	1.04×
	FASTCODER	**7.88**	**2.53**×	**10.83**	**2.38**×	**10.80**	**2.43**×	**19.02**	**2.52**×
		$4.0s^{\downarrow 6.2s}$ / 34.33^{-}		$5.5s^{\downarrow 7.7s}$ / 37.79^{-}		$5.5s^{\downarrow 7.9s}$ / 38.17^{-}		$9.7s^{\downarrow 14.8s}$ / 37.58^{-}	

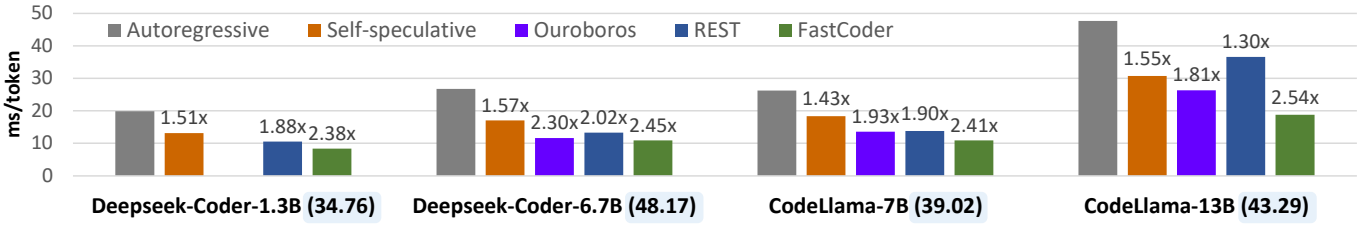


Fig. 5: Decoding speed and speedup ratio on standalone code generation dataset HumanEval. We also report the *Pass@1* results of FASTCODER in parentheses with a blue background, which are the same with autoregressive decoding.

model, we prioritize the smaller model from the same series as the target LLM. Thus, we choose Deepseek-Coder-base-1.3B for Deepseek-Coder-base-6.7B, and CodeLlama-Python-7B for CodeLlama-Python-13B as draft models. For CodeLlama-7B, which is the smallest model in its series, we opt for TinyLlama-1.1B as the draft model due to its shared architecture and tokenizer compatibility. Due to the small size of DeepSeek-Coder-1.3B, identifying a suitable draft model for it is challenging, and therefore, results for this setting are not reported. For REST, to construct the datastore, we select the first 10 files in Python subset of The Stack dataset [33], resulting in a datastore of approximately 8.7 GB in size. The values of the other hyper-parameters are consistent with those in the original paper.

VI. RESULTS AND ANALYSIS

A. RQ1: Overall Performance

To answer this research question, we compare the inference acceleration performance of FASTCODER against state-of-the-art approaches on both repository-level and standalone code generation tasks, and the detailed results and analysis are presented as follows.

1) Repository-level Code Generation: The comparison results between FASTCODER and baselines are shown in Table I. FASTCODER achieves up to 2.30× and 2.53× speedup on DevEval and RepoEval, respectively, outperforming state-of-the-art approaches by up to 88%. FASTCODER consistently maintains a stable speedup of more than 2× across a variety of backbone models and datasets, demonstrating its robustness. On average, FASTCODER saves 12.7s per sample on DevEval and 14.8s per sample on RepoEval compared with autoregressive decoding. The Pass@1 and Edit Similarity scores further confirm that FASTCODER produces sequences identical to those of autoregressive generation, as FASTCODER accepts draft tokens only when they match the target LLM's outputs.

Compared to the substantial speedups gained by FASTCODER, baseline approaches achieve limited accelerations. As a retrieval-based approach, the datastore utilized by REST is approximately 8 times the size of the one employed by FASTCODER, but the fixed datastore may suffer from misalignment between retrieval sequences and the LLM output. REST exhibits the optimal speedup of around 1.7× in most cases, but it performs poorly in experiments of CodeLlama-13B. This may be attributed to the fact that the significant CPU resource demands posed by both the 13B model inference and the retrieval of data from a large datastore in REST, leading to decreased performance. Besides, Ouroboros demonstrates comparable performance to REST on Deepseek-Coder-6.7B, yet its generation speed is even slower than autoregressive decoding on CodeLlama-7B, indicating that its efficacy is subject to considerable fluctuations influenced by factors such as model selection, further demonstrating the challenge of appropriate draft model selection. Self-speculative decoding consistently maintains a stable yet modest acceleration, and the preliminary search for skipped layers may introduce significant time overhead. In contrast, **FASTCODER does not require a**

draft model or additional training, yet it can maintain a stable speedup ratio even under resource-constrained conditions.

2) Standalone Code Generation: For FASTCODER, we remove D_r from the datastore and retain D_c, which is the same as the one used in the previous experiments. The results are shown in Fig. 5. **Even without the benefit of the multi-source datastore, FASTCODER still outperforms the baselines**, further demonstrating the effectiveness of the retrieval strategy and caching modules. The identical Pass@1 scores between FASTCODER and autoregressive decoding indicate that FASTCODER achieves acceleration without compromising model performance. Additionally, we observe that the baselines consistently perform better on HumanEval compared to repository-level datasets. This may be affected by the difference in difficulty between standalone and repository-level code generation tasks. For instance, Deepseek-Coder-1.3B achieves pass@1 scores of 34.8 on HumanEval and 18.2 on DevEval. Thus, for approaches such as Ouroboros and Self-speculative, which require a draft model, the performance in repository-level code generation may be negatively affected by the poor performance of the draft model. For REST, HumanEval involves no project-specific knowledge, and the common datastore may adequately satisfy retrieval requirements. The performance differences of existing approaches on the two types of code generation tasks also highlight that *evaluations based solely on standalone datasets may fail to reflect performance in real-world application scenarios.*

> **Answer to RQ1:** FASTCODER achieves state-of-the-art performance on both repository-level and standalone code generation tasks. It consistently maintains a stable speedup across a variety of backbone models and datasets, demonstrating its effectiveness and generalization.

B. RQ2: Ablation Study

To analyze the effectiveness of each component within FASTCODER, we conduct an ablation study. Since REST serves as a standard pipeline for retrieval-based draft-verification acceleration approach, we treat the results obtained using REST (with D_c as the datastore) as the baseline results. To ensure a thorough analysis, we first evaluate the performance of the baseline with each individual component added separately, followed by an evaluation of the baseline with any two components combined.

Table II reports the results for *decoding speed* and *average acceptance length*, demonstrating that each component contributes to a speedup gain. The multi-source datastore improves retrieval performance by offering richer and more interrelated content. This performance also represents REST using the same datastore ($D_r + D_c$) as FASTCODER, along with parallel retrieval. It indicates that the multi-source datastore design not only increases the average acceptance length but also reduces the external retrieval overhead through efficient parallel search mechanisms. The retrieval strategy optimizes the inference speed by eliminating unnecessary retrieval operations, which account for approximately 4% of the total retrievals. Crucially, this optimization has a negligible impact on the average acceptance length, ensuring minimal trade-off between efficiency and performance. Among all components, the CACHE mechanism proves to be the most effective, delivering a substantial 30%+ improvement in average acceptance length over the baseline. Statistical analysis shows that, although the CACHE contains only 174 sequences at most for DevEval, 33.13% of all retrieval operations can successfully obtain valid results directly from the CACHE. Besides, the average retrieval time from the cache is 0.2ms, which is approximately 15% of the retrieval time from the datastore D. This highlights the importance of alignment with the repository context and LLM preference. Furthermore, the combination of any two components consistently outperforms their individual effects, while integrating all three components achieves optimal performance, confirming positive interactions between components.

TABLE II: Ablation study results of FASTCODER on DevEval using Deepseek-Coder-6.7B. *AccLen* refers to average acceptance length.

	AccLen	**ms/token**	**Speedup**
Baseline	1.89	15.86	1.65×
+ datastore D ($D_r + D_c$)	2.28	14.82	1.76×
+ retrieval strategy	1.89	15.41	1.70×
+ CACHE	2.47	14.13	1.85×
+ datastore D & retrieval strategy	2.28	14.19	1.84×
+ retrieval strategy & CACHE	2.48	13.80	1.89×
+ datastore D & CACHE	2.85	11.94	2.19×
FASTCODER	**2.85**	**11.69**	**2.24×**

> **Answer to RQ2:** Each component of FASTCODER makes a contribution to the overall performance, with the CACHE mechanism proving to be the most effective, providing a substantial improvement in average acceptance length and speedup ratio.

C. RQ3: Integration with Correctness-Focused Approaches

To evaluate whether FASTCODER can benefit existing correctness-focused code generation approaches, we select RepoCoder [11] and RLCoder [18] for experiments, both of which follow RAG strategy. These approaches can be summarized as a two-phase process: ① retrieval of relevant code snippets from the repository through an optimized retrieval strategy to facilitate informative *prompt construction*, followed by ② *code generation* via LLM inference using the constructed prompt. Since FASTCODER solely accelerates LLM generation while preserving output consistency, it can be seamlessly integrated into the code generation phase without modifying the prompt construction stage from the original approach.

The speedup results can be seen in Table III, including the time spent on each phrase as well as the total execution time (seconds). As RepoCoder utilizes an iterative retrieval-generation pipeline, we adopt the optimal configuration (iterations = 2) from the original work and report the cumulative time consumption for each phase. From the results, we can

TABLE III: *Speedup* and *ES* results of integration of FASTCODER with existing correctness-focused code generation approaches on RepoEval using Deepseek-Coder-1.3B.

Approach	Time (s) Prompt	Generation	Total	Speedup	ES
RepoCoder	91.17	11,731.88	11,823.05	1.00×	30.35
+ FASTCODER		4,326.71	4,408.88	**2.68**×	30.35
RLCoder	204.11	5,419.50	5,623.61	1.00×	38.28
+ FASTCODER		1,908.78	2,112.89	**2.66**×	38.28

TABLE IV: Comparison of average acceptance length between FASTCODER and REST. *DE*, *RE*, and *HE* denote DevEval, RepoEval, and HumanEval respectively.

Bckbone Model	REST DE	RE	HE	FASTCODER DE	RE	HE
Deepseek-Coder-1.3B	2.04	2.04	2.38	2.97	3.21	2.87
Deepseek-Coder-6.7B	2.06	2.08	2.38	2.85	3.05	2.92
CodeLlama-7B	2.05	2.07	2.27	2.77	3.06	2.79
CodeLlama-13B	2.06	2.06	2.25	2.75	2.99	2.63

observe that, with prompt construction time held constant, FASTCODER substantially reduces the time required during the generation phase, achieving speedups of 2.68× and 2.66× on RepoCoder and RLCoder, respectively. To quantitatively assess code correctness, we also report Edit Similarity results in Table III, which demonstrates that FASTCODER and autoregressive decoding show consistent performance. This indicates high compatibility of FASTCODER, achieving inference speed acceleration through flexible integration with diverse existing code generation approaches following the RAG strategy while maintaining their correctness performance.

> **Answer to RQ3:** FASTCODER can be flexibly integrated with existing correctness-focused RAG-based code generation approaches, achieving a speedup of over 2.6× while preserving their original performance on correctness.

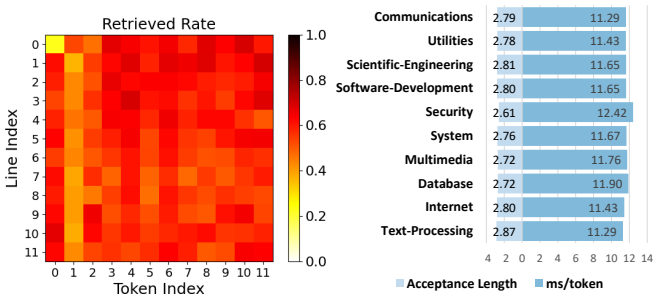
VII. DISCUSSION

A. Average Acceptance Length and Cost Analysis

As average acceptance length can reflect the theoretical upper bound of achievable acceleration for retrieval-based approaches, we compare the average acceptance length of FASTCODER with REST, the strongest retrieval-based baseline in most cases. The results are shown in Table IV. FASTCODER consistently exhibits a longer acceptance length across all datasets and backbone models, with an increase exceeding 50% compared on RepoEval. Specifically, the size of REST's datastore is approximately 8 times that of FASTCODER, but FASTCODER still achieves a higher acceleration upper bound, which demonstrates the importance of multi-source datastore as well as the context- and LLM preference-aware caching. Moreover, we further compare the resource consumption of FASTCODER with REST and autoregressive decoding, as reported in Table V. Since neither FASTCODER nor REST introduces an additional draft model, they incur no extra GPU overhead compared with autoregressive decoding. In comparison, FASTCODER shows markedly lower resource consumption than REST: its memory usage is less than one tenth, and its CPU usage is about one third, comparable to autoregressive decoding. With a compact yet effective datastore and accelerated construction speed, FASTCODER provides a more lightweight and efficient inference acceleration approach. In conclusion, FASTCODER can achieve a high average acceptance length, reflecting its ability to generate high-quality drafts, while also attaining a higher speedup ratio.

TABLE V: The resource consumption comparison on RepoEval using DeepSeek-Coder-6.7B.

Approach	GPU-VRAM	CPU	Memory
Autoregressive	21GB	114%	1.0GB
REST	21GB	335%	32.6GB
FASTCODER	21GB	128%	2.8GB



(a) (b)

Fig. 6: (a) Retrieval performance of FASTCODER; (b) Performance of FASTCODER on different code topics.

B. Heatmap of FASTCODER's Retrieval Performance

As mentioned in Section III-B, previous retrieval-based approaches often fail at the first non-whitespace token in each code line. To explicitly illustrate FASTCODER's effectiveness, we depict its retrieval performance heatmap in Fig. 6(a), with the evaluated samples and experimental settings aligned with Fig. 3(a). A clear observation is that Fig. 6(a) has a markedly darker color intensity, especially at the first non-whitespace token, indicating a significant increase in the probability of FASTCODER retrieving valid results. This improvement can be attributed to two factors: on the one hand, our retrieval strategy probabilistically avoids retrievals at such positions; on the other hand, the multi-source datastore and the context- and LLM preference-aware CACHE store high-quality sequences, which enhances the retrieval hit rate. Overall, this comparison underscores the enhanced retrieval efficacy of FASTCODER.

C. Performance on Different Topics

In practical software development, repositories are typically categorized into various domains, such as web development, text processing, *etc*. Each domain tends to exhibit distinct coding styles and conventions. Therefore, to investigate the generalization capacity of FASTCODER across different domains, we report Deepseek-Coder-6.7B results on DevEval's 10 topics. As shown in Fig. 6(b), FASTCODER demonstrates consistent and substantial acceleration across all topics, highlighting its robustness across diverse contexts.

Fig. 7: Case study of an example sourced from RepoEval, with legends positioned in the upper-right corner.

D. Case Study

To further illustrate the effectiveness of FASTCODER, we conduct a case study. As shown in Fig. 7, we use different background colors to highlight the sources of the accepted draft tokens utilizing FASTCODER. Additionally, the underlined tokens refer to those that can be retrieved by FASTCODER but not by the baseline (REST with D_c as the datastore). When generating the earlier parts, the CACHE remains unavailable due to an insufficient accumulation of sequences. Nonetheless, lots of repository-related tokens can be additionally retrieved by FASTCODER, benefiting from the multi-source datastore. When the CACHE is available, a larger number of consecutive tokens becomes retrievable, thereby enhancing the inference speed by extending acceptable sequences and reducing retrieval overhead.

E. Threats to Validity

Internal. The internal threat to validity lies in the potential risk of data leakage during the construction of the datastore D, as the code contained within the datastore D may overlap with the code that is intended to be generated. To mitigate this issue, we adopted precautionary measures during the construction of datastore D. Specifically, when building the repository-related datastore D_r, we excluded all portions requiring generation, thereby ensuring that D_r contains no ground-truth code. For the common datastore D_c, where potential leakage may occur due to the use of The Stack [33], we examined the collection timelines of all relevant resources: DevEval [17] (downloaded in November 2023), RepoEval [11] (repositories created after January 1, 2022), and The Stack (collected from January 2015 to June 2022). These timelines indicate that The Stack and DevEval share no temporal overlap, with only minor overlap possible with RepoEval. Moreover, only 1/145 of the Python subset of The Stack was utilized, thereby substantially reducing the likelihood of leakage. As previous studies [16] have also employed The Stack to construct datastores, we adopt a consistent practice in this work.

External. The external threat to validity stems from dataset and backbone model limitations, as we perform experiments on a limited set of datasets and models. Thus, the results obtained may not generalize to other experimental settings. To mitigate this limitation, we evaluate FASTCODER on three datasets spanning both repository-level and standalone code generation tasks, where FASTCODER consistently outperforms baseline approaches, confirming its generalization. Furthermore, FASTCODER maintains robust performance regardless of the underlying LLM, validated through systematic experiments with LLMs ranging from 1B to 13B parameters. Although we do not provide results on larger models due to resource constraints, FASTCODER's LLM preference-aware cache helps ensure that retrieved drafts remain effective by aligning them with the output style of the target LLM, thereby supporting generalization across different model scales.

VIII. CONCLUSION

In this paper, we propose FASTCODER, a simple yet efficient LLM inference acceleration approach for repository-level code generation without compromising generation quality. FASTCODER leverages a multi-source datastore as well as a context- and LLM preference- aware cache to improve the acceptance length of the retrieved draft, while minimizing redundant retrieval operations through a dynamic and efficient retrieval strategy. Experimental results demonstrate that FASTCODER outperforms state-of-the-art inference acceleration approaches on both repository-level and standalone code generation tasks. FASTCODER can also be integrated with correctness-focused code generation approaches to accelerate their inference speed. Requiring no draft model or additional training, FASTCODER provides a lightweight and practical solution for LLM inference acceleration in code generation.

ACKNOWLEDGMENT

This research is supported by the National Natural Science Foundation of China Grants Nos. 62302021 and 62332001. We thank Borui Zhang and Runlin Guo for their contributions to the demo development.

REFERENCES

[1] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

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

[3] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

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

[5] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[7] M. Liang, X. Xie, G. Zhang, X. Zheng, P. Di, H. Chen, C. Wang, G. Fan *et al.*, "Repofuse: Repository-level code completion with fused dual context," *arXiv preprint arXiv:2402.14323*, 2024.

[8] H. N. Phan, H. N. Phan, T. N. Nguyen, and N. D. Bui, "Repohyper: Better context retrieval is all you need for repository-level code completion," *CoRR*, 2024.

[9] W. Liu, A. Yu, D. Zan, B. Shen, W. Zhang, H. Zhao, Z. Jin, and Q. Wang, "Graphcoder: Enhancing repository-level code completion via coarse-to-fine retrieval based on code context graph," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 570–581.

[10] W. Cheng, Y. Wu, and W. Hu, "Dataflow-guided retrieval augmentation for repository-level code completion," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 7957–7977.

[11] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, "Repocoder: Repository-level code completion through iterative retrieval and generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 2471–2484.

[12] F. Liu, Z. Fu, G. Li, Z. Jin, H. Liu, Y. Hao, and L. Zhang, "Non-autoregressive line-level code completion," *ACM Transactions on Software Engineering and Methodology*, 2024.

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

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

[15] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, Z. Wang, Z. Zhang, R. Y. Y. Wong, A. Zhu, L. Yang, X. Shi *et al.*, "Specinfer: Accelerating large language model serving with tree-based speculative inference and verification," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 932–949.

[16] Z. He, Z. Zhong, T. Cai, J. Lee, and D. He, "Rest: Retrieval-based speculative decoding," in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2024, pp. 1582–1595.

[17] J. Li, G. Li, Y. Zhao, Y. Li, H. Liu, H. Zhu, L. Wang, K. Liu, Z. Fang, L. Wang *et al.*, "Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories," in *Findings of the Association for Computational Linguistics ACL 2024*, 2024, pp. 3603–3614.

[18] Y. Wang, Y. Wang, D. Guo, J. Chen, R. Zhang, Y. Ma, and Z. Zheng, "Rlcoder: Reinforcement learning for repository-level code completion," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 165–177.

[19] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[20] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[21] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[22] Y. Ding, Z. Wang, W. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth *et al.*, "Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[23] M. Ghazvininejad, O. Levy, Y. Liu, and L. Zettlemoyer, "Mask-predict: Parallel decoding of conditional masked language models," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 6112–6121. [Online]. Available: https://aclanthology.org/D19-1633/

[24] C. Chen, S. Borgeaud, G. Irving, J.-B. Lespiau, L. Sifre, and J. Jumper, "Accelerating large language model decoding with speculative sampling," *arXiv preprint arXiv:2302.01318*, 2023.

[25] Y. Leviathan, M. Kalman, and Y. Matias, "Fast inference from transformers via speculative decoding," in *International Conference on Machine Learning*. PMLR, 2023, pp. 19 274–19 286.

[26] M. Stern, N. Shazeer, and J. Uszkoreit, "Blockwise parallel decoding for deep autoregressive models," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2018/file/c4127b9194fe8562c64dc0f5bf2c93bc-Paper.pdf

[27] T. Cai, Y. Li, Z. Geng, H. Peng, J. D. Lee, D. Chen, and T. Dao, "Medusa: Simple llm inference acceleration framework with multiple decoding heads," in *Proceedings of the 41st International Conference on Machine Learning*, 2024, pp. 5209–5235.

[28] Y. Fu, P. Bailis, I. Stoica, and H. Zhang, "Break the sequential dependency of llm inference using lookahead decoding," in *International Conference on Machine Learning*, 2024.

[29] Y. Li, F. Wei, C. Zhang, and H. Zhang, "Eagle: Speculative sampling requires rethinking feature uncertainty," in *International Conference on Machine Learning*, 2024.

[30] N. Yang, T. Ge, L. Wang, B. Jiao, D. Jiang, L. Yang, R. Majumder, and F. Wei, "Inference with reference: Lossless acceleration of large language models," *arXiv preprint arXiv:2304.04487*, 2023.

[31] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[32] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 269–280.

[33] D. Kocetkov, R. Li, C. Mou, Y. Jernite, M. Mitchell, C. M. Ferrandis, S. Hughes, T. Wolf, D. Bahdanau, L. Von Werra *et al.*, "The stack: 3 tb of permissively licensed source code," *Transactions on Machine Learning Research*, 2022.

[34] B. F. Spector and C. Re, "Accelerating llm inference with staged speculative decoding," in *Workshop on Efficient Systems for Foundation Models@ ICML2023*, 2023.