CodeSwift: Accelerating LLM Inference for Efficient Code Generation

Anonymous ACL submission

Abstract

Code generation is a latency-sensitive task that demands high timeliness, but the autoregressive decoding mechanism of Large Language Models (LLMs) leads to poor inference efficiency. Existing LLM inference acceleration methods mainly focus on standalone functions using only built-in components. Moreover, they treat code like natural language sequences, ignoring its unique syntax and semantic characteristics. As a result, the effectiveness of these approaches in code generation tasks remains limited and fails to align with real-world programming scenarios. To alleviate this issue, we propose CodeSwift, a simple yet highly efficient inference acceleration approach specifically designed for code generation, without comprising the quality of the output. CodeSwift constructs a multi-source datastore, providing access to both general and project-specific knowledge, facilitating the retrieval of high-quality draft sequences. Moreover, CodeSwift reduces retrieval cost by controlling retrieval timing, and enhances efficiency through parallel retrieval and a context- and LLM preference-aware cache. Experimental results show that CodeSwift 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%. Our code and data are available at `https://anonymous.4open.science/r/CodeSwift`.

1 Introduction

Large Language Models (LLMs) such as GPT-4o (Achiam et al., 2023) and DeepSeek-Coder (Guo et al., 2024) have demonstrated impressive performance in coding tasks, revolutionizing the landscape of software development (Github, 2021; Li et al., 2023). These models excel in code completion and generation but face a challenge: the significant inference time. LLMs use the autoregressive decoding mechanism, where each new token is generated conditioned on the previously generated tokens and the given context. However, developers typically hold high expectations regarding the responsiveness of code recommendations (Liu et al., 2024a). If LLMs fail to deliver precise and efficient feedback, it may directly affect development efficiency and user experience.

To accelerate the inference process of LLMs, speculative decoding (Chen et al., 2023a; Leviathan et al., 2023) is regarded as one of the effective solutions, which employs a draft-verification framework to minimize the number of forward steps. Specifically, it utilizes a small language model as a draft model to rapidly generate candidate output tokens, which are then verified for acceptability by the target LLM through a single forward step while keeping the output consistent with that decoded autoregressively by the target LLM itself. Based on the draft-verification paradigm, many inference acceleration approaches have emerged (Chen et al., 2023b; Zhang et al., 2024; Zhao et al., 2024; Li et al., 2024b; Miao et al., 2024), most of which rely on an additional draft model, either selected from the same model family or trained for specific use cases. However, identifying a suitable draft model remains challenging, as it requires striking a delicate balance between maintaining a small model size and ensuring high output quality. Additionally, the draft model must align with the vocabulary of the target LLM, further complicating the selection process. More recently, researchers have explored replacing the parametric draft model with a non-parametric retrieval system (He et al., 2024; Yang et al., 2023), which can easily be ported to any LLM without introducing additional training costs and have be applied to code generation task.

While some of the above approaches have demonstrated promising performance in code generation task (He et al., 2024; Zhao et al., 2024), they primarily focus on standalone code functions

```python
def has_close_elements(numbers, threshold):
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
            if distance < threshold:
                return True
    return False
```

(a) A standalone function

```python
from lightweight_mmm.core import priors, core_utils
def sinusoidal_seasonality(...)-> jnp.ndarray:
    number_periods = data.shape[0]
    default_priors = priors.get_default_priors()
    n_geos = core_utils.get_number_geos(data=data)
    with numpyro.plate(f"{priors.GAMMA_SEASONALITY}_sin_cos_plate", 2):
        with numpyro.plate(f"{priors.GAMMA_SEASONALITY}_plate", ...):
            gamma_seasonality = numpyro.sample(
                name=priors.GAMMA_SEASONALITY,
                fn=custom_priors.get(
                    priors.GAMMA_SEASONALITY,
                    default_priors[priors.GAMMA_SEASONALITY]))
    ...
    seasonality_values = _sinusoidal_seasonality(
        seasonality_arange=seasonality_arange, ...)
    ...
    return seasonality_values
```

(b) A repository-level function

Figure 1: Examples of standalone and repository-level functions. Intra-file and cross-file dependencies are highlighted in green and yellow, respectively.

that solely rely on built-in components. However, in real-world software development, it is crucial for developers to be aware of other files within the repository during programming (Zhang et al., 2023), which gives rise to repository-level code generation (more details in Appendix A). As shown in Figure 1, complex dependencies that span multiple levels can exist in repository-level functions. *Experimental results show that existing inference acceleration approaches typically perform worse on repository-level code generation under the same settings than standalone ones.* For example, Self-speculative decoding (Zhang et al., 2024) can achieve over 1.5× acceleration compared to autoregressive decoding in standalone code generation (Figure 5), but falls short when applied to repository-level tasks, offering virtually no speedup in comparison to autoregressive inference (Table 1). Moreover, existing approaches treat source code as sequences similar to natural language, without accounting for code's unique syntactic and semantic characteristics. *As a result, the effects of existing LLM inference acceleration approaches on code generation tasks may be limited and fail to align with real-world scenarios.*

To alleviate this issue, in this paper, we primarily focus on improving the inference speed of LLMs on code generation task, covering both repository-level and standalone code, without comprising the quality of the output. We propose CODESWIFT, a **simple yet highly efficient** approach to accelerate the inference of LLMs through an efficient and effective retrieval strategy. Concretely, we first construct a multi-source datastore, providing access to both general and project-specific knowledge and enhancing the quality of draft sequences. Then, CODESWIFT reduces unnecessary retrieval overhead by controlling the retrieval timing. Besides, CODESWIFT improves retrieval efficiency through parallel retrieval and the maintenance of a context- and LLM preference-aware cache. Finally, tree attention is employed to avoid redundant computation caused by verifying multiple draft sequences. Experimental results show that the decoding speed of CODESWIFT surpasses existing inference acceleration approaches substantially on both repository-level and standalone code generation tasks. For repository-level code generation, CODESWIFT achieves up to 2.30× and 2.53× speedup on DevEval (Li et al., 2024a) and RepoEval (Zhang et al., 2023), respectively. CODESWIFT can also achieve up to 2.54× acceleration on standalone code generation dataset, HumanEval (Chen et al., 2021). It is worth noting that incorporating project-specific knowledge enables the generation of high-quality drafts, reducing the verification time and, consequently, the inference time of our model for repository-level code generation. However, this knowledge can be omitted in standalone code generation where such context is unnecessary.

Our contributions can be summarized as follows:

- We identify limitations of current LLM inference acceleration approaches within the context of real-world code generation and provide insights for potential improvements.
- We propose CODESWIFT, a simple yet efficient approach to accelerate LLM inference for code generation by leveraging effective retrieval and verification mechanisms.
- We conduct a comprehensive evaluation of CODESWIFT and results show that it achieves state-of-the-art results in both repository-level and standalone code generation tasks.

2 Related Work

Autoregressive decoding generates tokens sequentially, leading to slow and costly decoding. To accelerate this process, draft-verification approaches (Chen et al., 2023a; Miao et al., 2024; He et al., 2024) have gained popularity recently as they enhance speed without compromising performance, which fall into generation-based and retrieval-based categories based on their draft generation techniques (more information in Appendix B).

Generation-based approaches. Draft tokens can be generated either by a smaller model or by the target model itself. Speculative decoding (Chen et al., 2023a; Leviathan et al., 2023) employs a smaller model for drafting and uses the target LLM for efficient parallel verification. Ouroboros (Zhao et al., 2024) generates draft phrases to enhance parallelism and extend drafts. Alternatively, the target LLM itself can be utilized to efficiently draft (Stern et al., 2018; Li et al., 2024b; Fu et al., 2024), which reduces system complexity and selection difficulties. Medusa (Cai et al., 2024) introduces multiple heads to predict multiple draft tokens in parallel. Self-speculative decoding (Zhang et al., 2024) employs the target model with selectively certain intermediate layers skipped as the draft model.

Retrieval-based approaches. The retrieval-based draft generation approach replaces the model generation with a search in a retrieval datastore to obtain candidate sequences. These approaches avoid extra training and can reduce computational overhead. LLMA (Yang et al., 2023) is an inference-with-reference decoding mechanism by exploiting the overlap between the output and the reference of an LLM. REST (He et al., 2024) replaces the parametric draft model with a non-parametric retrieval datastore.

3 Preliminaries

3.1 Retrieval-based Speculative Decoding

Building upon the draft-verification framework introduced by speculative decoding (Chen et al., 2023a; Leviathan et al., 2023), retrieval-based decoding acceleration approaches leverage a retrieval mechanism to generate draft tokens (He et al., 2024; Yang et al., 2023), 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) (He et al., 2024), which has proven to be effective in standalone function generation task (Chen et al., 2021). Below is an explanation of how it works. Pre-built from a code corpus, the datastore of $D = \{(c_i, t_i)\}$ 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 (Manber and Myers, 1993). Specifically, given a context $s = (x_1, ..., x_t)$, it aims to

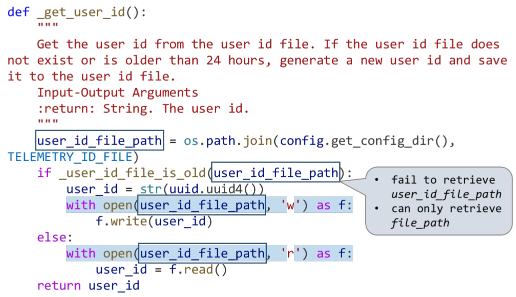


Figure 2: Localness of source code.

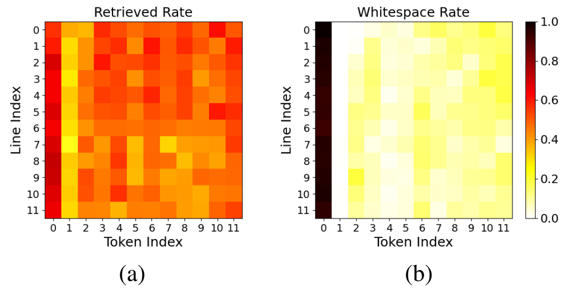


Figure 3: Heatmaps of (a) retrieval performance and (b) whitespace distribution with token positions in REST. The maximum token index is selected based on the average token number per line (12).

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, CODESWIFT also incorporates a similar suffix-match-based retrieval algorithm, leveraging its advantages in time and memory efficiency.

3.2 Motivating Examples

To identify the limitations of current inference acceleration methods in code generation, 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 (Tu et al., 2014), 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 Figure 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 code 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 (Li et al., 2024a), a repository-level code generation benchmark, and employed DeepSeek-Coder-6.7B (Guo et al., 2024) 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 Figure 3. As seen from Figure 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.*

4 Method

The architecture of CODESWIFT is shown in Figure 4. In this section, we first describe the construction of datastore and cache, and then provide a detailed explanation of retrieval and verification process.

4.1 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 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, CODESWIFT 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) 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 (Kocetkov et al., 2022) 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 He et al. (2024). 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 CODESWIFT 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). For standalone code generation, D_r can be omitted.

4.2 Context- and LLM Preference-aware Caching

To reduce retrieval costs and improve the alignment of retrieved results with 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 se-

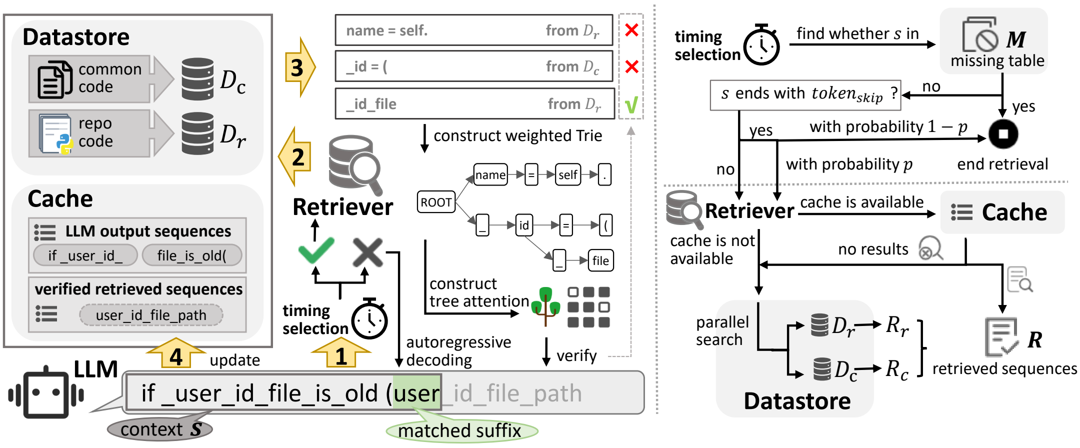


Figure 4: Architecture of CODESWIFT. The left part illustrates an overview, and the right part offers a detailed depiction of timing selection and retrieval operation.

quences. Specifically, based on the observations in Section 3.2, program entities (token sequences) defined or used in preceding snippets are often reused in the subsequent code snippets. 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. Moreover, since the datastore D is static, the draft sequences retrieved for the identical context s remain consistent. However, different LLMs exhibit distinct generation preferences, leading to varied decoding outputs after draft verification. Additionally, earlier decoding outputs must maintain contextual relevance and coherence with subsequent outputs. 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 predefined threshold l, it is accessible and will remain active throughout the entire inference process.

4.3 Dynamic and Efficient Retrieval Strategy

Algorithm 1 illustrates the complete retrieval process of CODESWIFT. Before each forward step, given current context s, CODESWIFT 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, CODESWIFT utilizes a dynamic and efficient retrieval strategy to minimize unnecessary retrieval cost. Specifically, CODESWIFT optimizes retrieval timing by addressing two key considerations as follows.

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.$size \geq l$ **then**
    // retrieval from cache
    $R \leftarrow$ search(CACHE)
if CACHE.$size < l$ ***or*** $R = \emptyset$ **then**
    // retrieval timing selection
    if $s \in M$ **then**
        pass
    else if *s ends with* $token_{skip}$ **then**
        if *random number* $< p$ **then**
            // parallel retrieval from datastore
            $R_r, R_c \leftarrow$ par_search(D_r, D_c)
            $R \leftarrow (R_r, R_c)$
if $R = \emptyset$ **then**
    // update missing table
    $M \leftarrow M \cup \{s\}$
else
    update CACHE
return R;

Skip token. As mentioned in Section 3.2, 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 token 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, CODESWIFT will bypass the retrieval and directly utilize the LLM to generate the next token.

If CODESWIFT decides to proceed with retrieval according to the above strategy, parallel retrieval is conducted from repository-related datastore D_r and common datastore 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.

4.4 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. By controlling the values of α and β, the preference of draft sequences can be adjusted to accommodate different scenarios. We select top-k sequences from the Trie, ordered by their weights from highest to lowest, as the draft sequences. Subsequently, the draft sequences are verified by LLM using tree attention (Spector and Re, 2023; Miao et al., 2024). 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.

5 Experiments

5.1 Experimental Setup

Datasets. We conduct experiments on both repository-level and standalone code generation benchmarks. For repository-level code generation, we choose two widely-used benchmarks, DevEval (Li et al., 2024a) and RepoEval (Zhang et al., 2023). 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 (Chen et al., 2021), a widely-used standalone code generation dataset including 164 human-written programming problems.

Backbone Models. We use the 1.3B and 6.7B configurations of Deepseek-Coder-base (Guo et al., 2024), as well as 7B and 13B configurations of CodeLlama-Python (Roziere et al., 2023) for evaluation, which are popular and well-performing LLMs in code generation.

Baselines. We compare CODESWIFT with vanilla autoregressive decoding and several state-of-the-art inference acceleration approaches that follow the draft-verification framework and have demonstrated effectiveness in code generation, including Self-speculative decoding (Zhang et al., 2024), Ouroboros (Zhao et al., 2024), and REST (He et al., 2024). Self-speculative decoding requires several hours to identify skipped layers in the target LLM for draft model construction. Ouroboros demands manual selection of a suitable draft model for the target LLM. REST is draft model-free but suffers from misalignment between retrieval sequences and the LLM output.

Evaluation Metrics. We report the *decoding speed* (ms/token) and the *speedup ratio* compared with vanilla autoregressive decoding. We also compare the *average acceptance length*, defined as the average number of tokens accepted per forward step by the target LLM, which reflects the upper bound of achievable acceleration. Since CODESWIFT and baselines do not compromise model performance, the correctness of the generated code is not evaluated.

Implementation Details. To provide essential contextual information, we prepend preceding code snippets from the same file as context for DevEval and RepoEval. All results are obtained with a maximum input length of 2k and a maximum generation length of 512 under greedy decoding. We focus on greedy decoding results as baseline approaches perform optimally with greedy decoding and comparably to other sampling strategies. D_c is constructed from a subset of Python pre-training code in The Stack (Kocetkov et al., 2022), taking approximately

Table 1: Decoding speed and speedup ratio on repository-level code generation datasets.

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×
	CODESWIFT	**8.71**	**2.30×**	**11.69**	**2.24×**	**12.17**	**2.16×**	**21.56**	**2.15×**
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×
	CODESWIFT	**7.88**	**2.53×**	**10.83**	**2.38×**	**10.80**	**2.43×**	**19.02**	**2.52×**

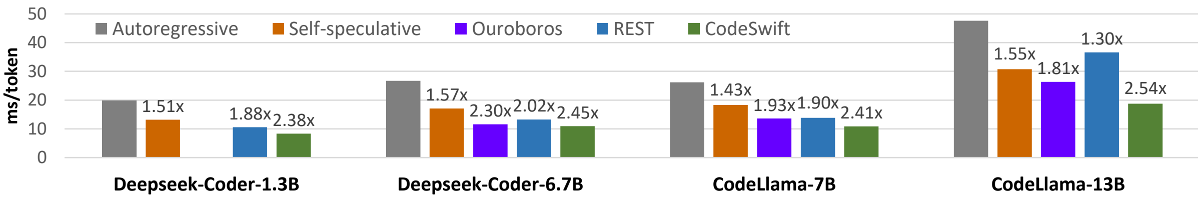


Figure 5: Decoding speed and speedup ratio on HumanEval.

9 minutes and yielding a 0.9GB datastore. D_r ranges from 60KB and 289MB across repositories, taking an average of 10 seconds. Hyper-parameters include $l = 50$, $p = 0.5$, $\alpha = \beta = 1$, with LLM output truncated every 20 tokens and added to the CACHE. Following He et al. (2024), 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. Baseline implementation details are in Appendix C. Experiments for Deepseep-Coder 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.

5.2 Main Results

5.2.1 Repository-level Code Generation

The comparison results between CODESWIFT and baselines are shown in Table 1. CODESWIFT achieves up to 2.30× and 2.53× speedup on DevEval and RepoEval, respectively, outperforming state-of-the-art approaches by up to 88%. CODESWIFT consistently maintains a stable speedup of more than 2× across a variety of backbone models and datasets, and repositories spanning various topics (Appendix D), demonstrating its robustness.

Compared to the substantial speedups gained by CODESWIFT, 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 CODESWIFT. 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. Self-speculative decoding consistently maintains a stable yet modest acceleration. On the contrast, **CODESWIFT does not require a draft model or additional training, yet it can maintain a stable speedup ratio even under resource-constrained conditions.**

5.2.2 Standalone Code Generation

For CODESWIFT, 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 Figure 5. **Even without the benefit of the multi-source datastore, CODESWIFT still outperforms the baselines**, further demonstrating the effectiveness of the retrieval strategy and caching modules. Additionally, we observe that the baselines consistently perform better on HumanEval compared to repository-level datasets. This may be affected by the difficulty difference 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

Table 2: Ablation study results of CODESWIFT on DevEval using Deepseek-Coder-6.7B. Each component is incrementally added. The baseline results are obtained using REST with D_c as the datastore. *AccLen* refers to average acceptance length.

	AccLen	ms/token	Speedup
Baseline	1.89	15.86	1.65×
+ multi-source datastore	2.28	14.82	1.76×
+ retrieval strategy	2.28	14.19	1.84×
+ CACHE	2.85	11.69	2.24×

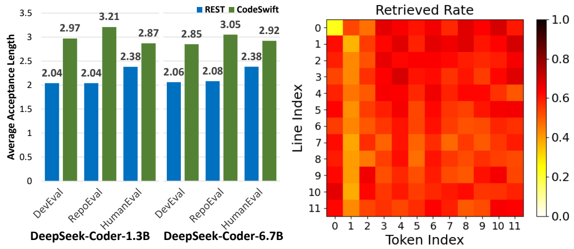
DevEval. Thus, for approaches such as Ouroboros and Self-speculative which require a draft model, the performance in repository-level code generation may be negatively affected by the poor performance of the draft model. For REST, HumanEval involves no project-specific knowledge, and the common datastore 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.*

5.3 Ablation Study

To analyze the effectiveness of each component within CODESWIFT, we conduct an ablation study with the results presented in Table 2. Each component is found to contribute to a speedup gain. The multi-source datastore provides richer and more interrelated retrieval content, not only enhancing the average acceptance length but also minimizing the external retrieval cost through parallel search. The retrieval strategy accelerates the inference by reducing unnecessary retrieval operations (4.02% of the total count of retrieval), with negligible impact on the average acceptance length. The CACHE is the most effective component, which provides an additional increase in average acceptance length of over 30% compared to 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. The average retrieval time from the cache is 0.2ms, which is approximately 15% of the retrieval time from the datastore. A case study is shown in Appendix F.

5.4 Analysis of Acceptance Length

We compare the acceptance length between CODESWIFT and REST (the best performing baseline), which represents the upper bound of achievable acceleration. The results is shown in Figure 6(a) (more results in Appendix E). **CODESWIFT exhibits a longer acceptance length across all datasets, with an increase exceeding 50% compared to REST on RepoEval.** Although the size of REST's datastore is approximately 8 times that of CODESWIFT, CODESWIFT achieves a higher acceleration upper bound. As REST's performance improves with the increasing size of the datastore when resources are sufficient (He et al., 2024), we do not claim that CODESWIFT can outperform REST under all circumstances. Nonetheless, CODESWIFT provides a more lightweight and efficient inference acceleration approach.



(a) (b)

Figure 6: (a) Acceptance length of CODESWIFT and REST; (b) Retrieval performance of CODESWIFT.

5.5 Heatmap of Retrieval Performance

To explicitly illustrate CODESWIFT's effectiveness, we depict its retrieval performance heatmap in Figure 6(b), with all settings aligned with Figure 3(a). A clear observation is that the overall color intensity of Figure 6(b) is markedly darker compared to Figure 3(a), indicating a significant increase in the probability of CODESWIFT retrieving valid results. This improvement underscores the enhanced retrieval efficacy of CODESWIFT.

6 Conclusion

In this paper, we propose CODESWIFT, a simple and efficient LLM inference acceleration approach for code generation without compromising generation quality. CODESWIFT leverages a multi-source datastore and 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 CODESWIFT outperforms state-of-the-art inference approaches in decoding speed for both standalone and repository-level code generation tasks. Requiring no draft model or additional training, CODESWIFT provides a lightweight and practical solution for LLM inference acceleration in code generation.

Limitations

Although CODESWIFT offers advantages in accelerating LLM inference for code generation, it also has limitations that need to be taken into account. Firstly, we only present the experimental results on code generation benchmarks written in Python. Nevertheless, CODESWIFT is designed to be universally applicable and can be seamlessly extended to other programming languages. Additionally, in the process of integrating repository code into the datastore, CODESWIFT directly utilizes the entire code files. However, the development of an effective method for extracting high-frequency expressions from repositories could potentially enhance performance.

Ethical Considerations

We emphasize that the entirety of our research is based on open-source datasets, models, and tools. Our method has no potential risk since it is training-free and has no impact on the generation results.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

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

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and et al. 2021. Evaluating large language models trained on code.

Ziyi Chen, Xiaocong Yang, Jiacheng Lin, Chenkai Sun, Kevin Chen-Chuan Chang, and Jie Huang. 2023b. Cascade speculative drafting for even faster llm inference. *arXiv preprint arXiv:2312.11462*.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2024. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36.

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. 2024. Break the sequential dependency of llm inference using lookahead decoding. In *International Conference on Machine Learning*.

Marjan Ghazvininejad, Omer Levy, Yinhan Liu, and Luke Zettlemoyer. 2019. 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)*, pages 6112–6121, Hong Kong, China. Association for Computational Linguistics.

Github. 2021. Github copilot.

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

Zhenyu He, Zexuan Zhong, Tianle Cai, Jason Lee, and Di He. 2024. 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)*, pages 1582–1595.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. 2024a. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. In *ACL (Findings)*, pages 3603–3614. Association for Computational Linguistics.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

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

Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, Hongwei Chen, Chengpeng Wang, Gang Fan, et al. 2024. Repofuse: Repository-level code completion with fused dual context. *arXiv preprint arXiv:2402.14323*.

Fang Liu, Zhiyi Fu, Ge Li, Zhi Jin, Hui Liu, Yiyang Hao, and Li Zhang. 2024a. Non-autoregressive line-level code completion. *ACM Transactions on Software Engineering and Methodology*.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2024b. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*.

Udi Manber and Gene Myers. 1993. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. 2024. 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*, pages 932–949.

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

Benjamin Frederick Spector and Christopher Re. 2023. Accelerating llm inference with staged speculative decoding. In *Workshop on Efficient Systems for Foundation Models@ ICML2023*.

Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. 2018. Blockwise parallel decoding for deep autoregressive models. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective retrieval for repository-level code completion. In *International Conference on Machine Learning*.

Nan Yang, Tao Ge, Liang Wang, Binxing Jiao, Daxin Jiang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023. Inference with reference: Lossless acceleration of large language models. *arXiv preprint arXiv:2304.04487*.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*.

Jun Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. 2024. 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)*, pages 11263 – 11282.

Weilin Zhao, Yuxiang Huang, Xu Han, Wang Xu, Chaojun Xiao, Xinrong Zhang, Yewei Fang, Kaihuo Zhang, Zhiyuan Liu, and Maosong Sun. 2024. Ouroboros: Generating longer drafts phrase by phrase for faster speculative decoding. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 13378–13393.

A Repository-level Code Generation

Code generation refers to the generation of code snippets that meet requirements based on natural language requirements. Most previous researches, such as the widely used datasets HumanEval(Chen et al., 2021) and MBPP (Austin et al., 2021), focus on standalone scenarios, which means the generated functions may invoke or access only built-in functions and standard libraries.

Researches on standalone code generation often diverges from the complexities of real-world programming tasks. In practical development settings, developers typically work within project environments, where the code to be generated is frequently intertwined with external contexts, such as API calls. This code often relies on the methods and properties defined in other files. These non-standalone functions constitute more than 70% of the functions in popular open-source projects, and evaluating models' effectiveness on standalone functions cannot reflect these models' effectiveness on pragmatic code generation scenarios (i.e., code generation for real settings of open source or proprietary code) (Yu et al., 2024). Consequently, there has been growing interest in repository-level code generation (Liu et al., 2024b; Wu et al., 2024; Liang et al., 2024), which refers to leveraging repository-level context during code generation tasks, rather than restricting the context to the file under development. Code files within a repository are often interdependent, featuring cross-module API calls, shared global snippets, and other forms of linkage. Researchers have introduced benchmark datasets such as RepoEval (Zhang et al., 2023), CoderEval (Yu et al., 2024), CrossCodeEval (Ding et al., 2024) and DevEval (Li et al., 2024a). These datasets provide structured means for assessing the quality and relevance of generated code in realistic scenarios.

B LLM inference acceleration approaches

Autoregressive decoding generates tokens in a step-by-step manner and results in a slow and costly decoding process. In order to accelerate decoding, non-autoregressive decoding approaches (Ghazvininejad et al., 2019; Liu et al., 2024a) that can generate multiple tokens in parallel have been proposed. While improving decoding speed, these approaches typically affect the model performance. Therefore, draft-verification decoding acceleration approaches (Chen et al., 2023a; Miao et al., 2024; He et al., 2024) have been widely adopted recently, which do not comprise the model performance. These approaches can be further categorized into generation-based and retrieval-based, depending on the technique used for draft generation.

B.1 Generation-based Approaches

The draft token can be generated either by the target LLM itself or by a small model. Using the target LLM itself to directly generate the token may get a higher acceptance rate, while using a small model is more likely to have a faster generation speed.

Using a small model. Speculative decoding (Chen et al., 2023a; Leviathan et al., 2023) is one of the effective acceleration approaches that minimize the target LLM forward steps by using an smaller model for drafting and then employing the target LLM to verify the draft in a low-cost parallel manner. Ouroboros (Zhao et al., 2024) generates draft phrases to parallelize the drafting process and lengthen drafts. Specinfer (Miao et al., 2024) uses many draft models obtained from distillation, quantization, and pruning to conduct speculations together.

Using the target LLM itself. 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 (Stern et al., 2018) installs multiple heads on the transformer decoder, enabling parallel generation of multiple tokens per step. Medusa (Cai et al., 2024) introduces multiple heads to predict multiple draft tokens in parallel. Lookahead decoding (Fu et al., 2024) uses a n-gram pool to cache the historical n-grams generated so far. Eagle (Li et al., 2024b) conducts the drafting process at the more structured feature level. Self-speculative decoding (Zhang et al., 2024)) employs the target LLM with selectively certain intermediate layers skipped as the draft model.

B.2 Retrieval-based Approaches

The retrieval-based draft generation approach replaces the model generation with a search in a retrieval datastore to obtain candidate sequences. These approaches can avoid extra training and reduce computational overhead. LLMA (Yang

Table 3: The skipped layers utilized in draft models for Self-speculative decoding.

	Index of Skipped Attention Layers	Index of Skipped MLP Layers
Deepseek-Coder-1.3B	[3, 6, 8, 9, 10, 13, 14, 15, 16, 18, 21, 22]	[4, 6, 9, 10, 20]
Deepseek-Coder-6.7B	[2, 5, 7, 8, 11, 12, 16, 18, 19, 20, 22, 23, 24, 25, 26, 28]	[2, 5, 6, 12, 15, 25, 26, 27, 28]
CodeLlama-7B	[4, 5, 7, 10, 11, 12, 13, 14, 18, 20, 21, 22, 27, 29, 31]	[8, 11, 13, 22, 23, 25, 27, 28, 31]
CodeLlama-13B	[5, 6, 9, 10, 11, 14, 15, 16, 21, 23, 24, 26, 27, 28, 29, 30, 31, 32, 34, 35, 36, 37]	[10, 11, 12, 14, 15, 25, 26, 27, 30, 32, 33, 34]

et al., 2023) 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 (He et al., 2024) 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.

C Implementation Details of Baselines

Self-speculative decoding. For the selection of skipped layers, we adopt the results provided by the authors (Zhang et al., 2024) for CodeLlama-13B. As for DeepSeek-Coder and CodeLlama-7B, for which the authors did not provide skipped layer configurations, we utilize Bayesian optimization on 4 samples from The Stack (Kocetkov et al., 2022) to determine the layers to skip during the drafting stage. The results can be seen in Table 3. Other settings remain consistent with the original paper.

Ouroboros. This approach requires a draft model for the target LLM, and our selection is illustrated in Table 4. We prioritize the selection of a smaller model from the same series as the target LLM to serve as the draft model. 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. For the configuration of hyper-parameters, we used $\gamma = 11$ for DeepSeek-Coder and $\gamma = 4$ for CodeLlama, following the recommendations provided in the original paper.

Table 4: Draft model selection for Ouroboros.

Target Model	Draft Model
Deepseek-Coder-base-6.7B	Deepseek-Coder-base-1.3B
CodeLlama-Python-7B	TinyLlama-1.1B-v1_math_code
CodeLlama-Python-13B	CodeLlama-Python-7B

REST. To construct the datastore, we select the first 10 files out of the 145 files in The Stack dataset (Kocetkov et al., 2022), resulting in a datastore of approximately 8.7 GB in size. The results of REST demonstrate that its performance on the HumanEval dataset improves as the size of the datastore increases (He et al., 2024). However, due to hardware limitations, we have chosen the largest feasible datastore that could be operated under the given constraints. The values of the other hyper-parameters are consistent with those in the original paper. Specifically, when performing exact match in the datastore, the starting context suffix length, n_{max}, is set to 16. The maximum number of selected draft tokens in the constructed Trie is set to 64.

D Performance on Different Code Topics

As DevEval includes code repositories spanning 10 distinct topics, we present the results of CodeSwift using Deepseek-Coder-6.7B for code generation separately for each topic. As shown in Figure 7, CodeSwift demonstrates consistent and substantial acceleration in code generation across all topics, highlighting its robustness and effectiveness in diverse contexts.

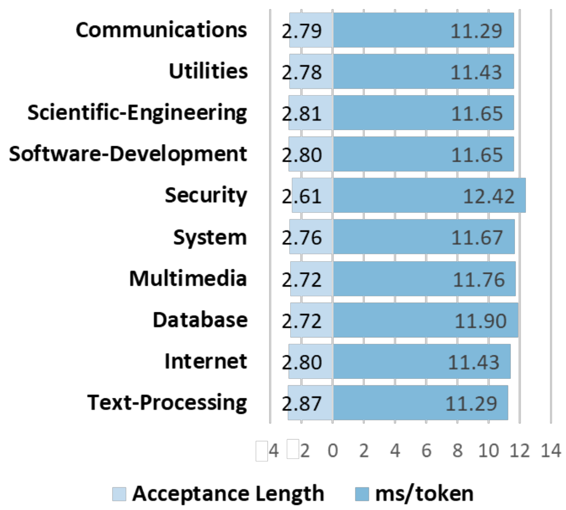


Figure 7: Performance of CodeSwift on different code topics.

E Comparison of Acceptance Length

We compare the acceptance length between CodeSwift and REST on both DeepSeek-Coder and CodeLlama. The results are shown in Table 5. CodeSwift exhibits a longer acceptance length across all datasets and backbone models.

Prompt

```python
import functools
from typing import Any, Dict, List, MutableMapping, Tuple, Union

import numpy as np
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.nn.functional as F
import transformers

try:
    from opendelta import (
        AdapterModel,
        BitFitModel,
        LoraModel,
        PrefixModel,
        SoftPromptModel,
    )

    HAS_OPENDELTA = True
except ModuleNotFoundError:
    HAS_OPENDELTA = False


def make_head(n_embd: int, out: int, dtype: type = torch.float32) -> nn.Sequential:
    """Returns a generic sequential MLP head."""
```

LLM output

tokens retrieved from datastore
tokens retrieved from cache
tokens that cannot be retrieved by baseline

```python
        return nn.Sequential(
        nn.Linear(n_embd, n_embd, dtype=dtype),
        nn.GELU(),
        nn.Linear(n_embd, out, dtype=dtype),
    )


def make_head_with_dropout(
    n_embd: int, out: int, dropout: float, dtype: type = torch.float32
) -> nn.Sequential:
    """Returns a generic sequential MLP head with dropout."""
    return nn.Sequential(
        nn.Linear(n_embd, n_embd, dtype=dtype),
        nn.GELU(),
        nn.Dropout(dropout),
        nn.Linear(n_embd, out, dtype=dtype),
    )


def make_head_with_dropout_and_layer_norm(
    n_embd: int, out: int, dropout: float, dtype: type = torch.float32
) -> nn.Sequential:
    """Returns a generic sequential MLP head with dropout and layer norm."""
    return nn.Sequential(
        nn.LayerNorm(n_embd, dtype=dtype),
        nn.Linear(n_embd, n_embd, dtype=dtype),
        nn.GELU(),
        nn.Dropout(dropout),
        nn.Linear(n_embd, out, dtype=dtype),
    )
```

Figure 8: Case study of CODESWIFT's retrieval performance.

Table 5: Acceptance length comparison between CODESWIFT and REST. *DC* and *CL* are abbreviations for Deepseek-Coder and CodeLlama, respectively.

	DevEval		RepoEval		HumanEval	
	REST	CODESWIFT	REST	CODESWIFT	REST	CODESWIFT
DC-1.3B	2.04	2.97	2.04	3.21	2.38	2.87
DC-6.7B	2.06	2.85	2.08	3.05	2.38	2.92
CL-7B	2.05	2.77	2.07	3.06	2.27	2.79
CL-13B	2.06	2.75	2.06	2.99	2.25	2.63

F Case Study

To demonstrate the effectiveness of CODESWIFT, we conduct a case study. As shown in Figure 8, we use different background colors to highlight the sources of the accepted draft tokens. Additionally, the tokens enclosed in red boxes are those that can be retrieved by CODESWIFT but not by the baseline (REST with D_c as the datastore). When generating the earlier parts of the sequence, the CACHE remains unavailable due to an insufficient accumulation of sequences. Nonetheless, lots of repository-related tokens can be additionally retrieved by CODESWIFT benefiting from the multi-source datastore. When the CACHE is available, a larger number of consecutive tokens becomes retrievable, thereby enhancing the inference speed through the extension of acceptable sequence lengths and the reduction of retrieval overhead.