From Logical to Computational Sparsity: Structure-Aware Block-Sparse Attention for Long Code Completion

Anonymous ACL submission

Abstract

Code Large Language Models face critical Time-To-First-Token (TTFT) latency challenges when handling long code completion due to the quadratic complexity ($O(n^2)$) of attention mechanisms. While existing sparse attention methods attempt to address this issue, they suffer from three key limitations: (1) general sparse patterns cause excessive accuracy degradation without considering code structure, (2) code-specific methods achieve only logical sparsity without actual computational speedup, and (3) limited adaptation to complex scenarios such as repository-level completion. We propose **SabreCoder**, a training-free **S**tructure-**a**ware **b**lock-spa**r**s**e** attention mechanism that bridges the gap between logical and computational sparsity. SabreCoder parses code into semantic chunks, constructs chunk-level sparse patterns through dependency analysis and similarity matching, and maps them to GPU-friendly block-sparse formats. Extensive experiments on LCC and CrossCodeEval benchmarks demonstrate that SabreCoder reduces TTFT by 45-55% while maintaining accuracy within 3% of dense attention.

1 Introduction

Code Large Language Models (Code LLMs) have demonstrated remarkable capabilities in code completion tasks (Guo et al., 2024; Lozhkov et al., 2024; Hui et al., 2024). They are now widely deployed as intelligent coding assistants in modern IDEs. However, when handling long code completion scenarios (Guo et al., 2023; Ding et al., 2023), Code LLMs face a critical challenge: the quadratic computational complexity ($O(n^2)$) of attention mechanisms leads to high Time-To-First-Token (TTFT) latency. This severely degrades the developer experience during interactive coding sessions.

To address the long-context generation latency issue, researchers have explored various approaches. These include model quantization, speculative decoding, and sparse attention mechanisms. Among these, sparse attention reduces computational cost by selectively attending to a subset of tokens, thereby lowering TTFT. However, existing sparse attention methods for long-code completion still suffer from three critical limitations.

Gap 1: General sparse patterns cause excessive accuracy drop. Prior general sparse attention methods (Xiao et al., 2023; Zaheer et al., 2020; Jiang et al., 2024) apply universal sparsity patterns. These include sliding windows, statistical token selection, random sampling, or importance-based filtering. While these methods can reduce TTFT, they do not consider the structural semantics of code. This structure-agnostic approach leads to excessive accuracy degradation.

Gap 2: Code-specific sparse patterns fail to achieve computational sparsity. Some prior works have explored code-specific sparse attention for BERT-based pre-training models (Guo et al., 2023; Wang et al., 2024). For instance, LongCoder treats import statements as globally visible and uses fixed-distance bridges to aggregate long-range dependencies. SparseCoder applies sparsity based on identifier patterns. Although these approaches achieve superior accuracy compared to general methods, they only realize *logical sparsity* rather than *computational sparsity*. The irregular, scattered nature of their attention patterns (e.g., bridges and identifiers dispersed across many blocks) prevents efficient mapping to hardware-friendly block-sparse kernels. This results in limited actual speedup despite a theoretical reduction in FLOPs.

Gap 3: Limited adaptation to complex code completion scenarios. Research has primarily focused on single-file code completion (Guo et al., 2023). However, with the advancement of Code LLMs, repository-level code completion (Ding et al., 2023) has emerged as a critical research

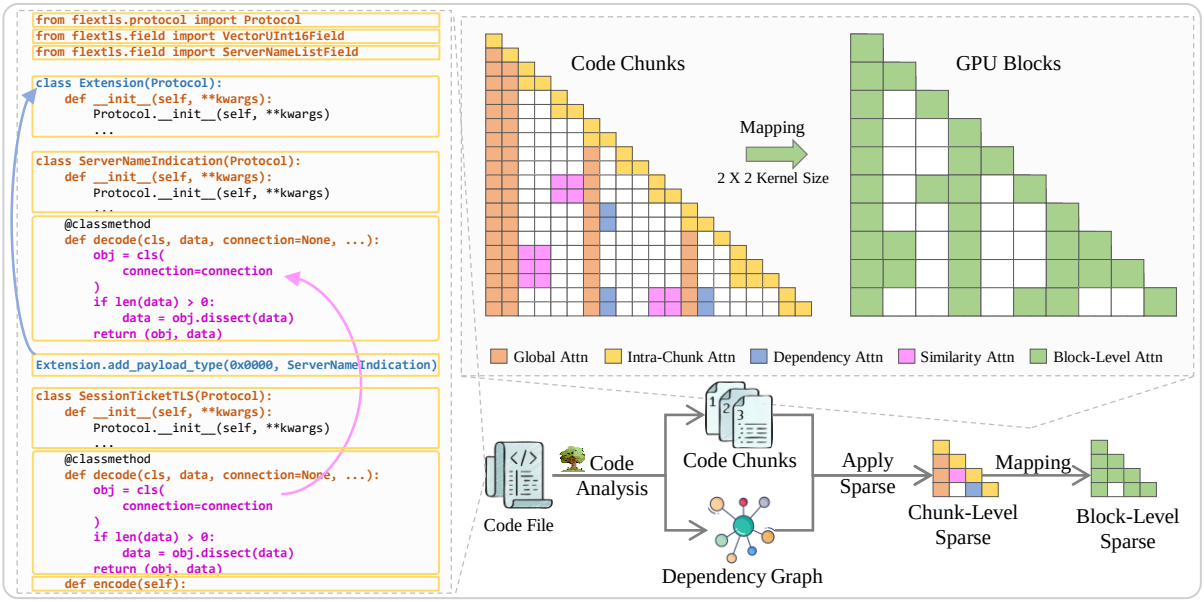

Figure 1: Overview of SabreCoder.

frontier. This scenario is more challenging as it requires modeling cross-file dependencies and repository-wide context. Existing sparse attention methods lack proper modeling for repository-level code completion scenarios, which often rely on the retrieval-augmented generation (RAG) paradigm. This leads to unacceptable accuracy degradation after sparsification.

To address these gaps, we propose **SabreCoder**, a training-free **S**tructure-**a**ware **b**lock-spa**r**s**e** attention mechanism. SabreCoder preserves code structural semantics while achieving genuine computational sparsity. By bridging the gap between logical and computational sparsity, SabreCoder significantly reduces TTFT with limited accuracy degradation.

The workflow of SabreCoder consists of three key stages: (1) **Code chunking and dependency parsing.** Given a code prefix, we divide the code into semantic chunks, where each chunk is a syntactic unit like a function, import statement, or code block. We then extract dependencies between these chunks. For RAG-based repository-level code completion, we treat each reference code snippet as a chunk. (2) **Chunk-level sparse pattern construction.** We apply four complementary sparse patterns to selectively preserve critical code relationships. These include intra-chunk attention, dependency-based attention, similarity-based attention, and global attention. (3) **Block-level sparse mapping.** We map the chunk-level sparse pattern to GPU-friendly block-sparse format. We implement custom Triton kernels to achieve genuine hardware acceleration.

The time complexity of our method is approximately $O(knL)$ where k is the average number of dependencies per chunk and L is the average chunk size. In practice, $kL \ll n$. This complexity is comparable to general sparse attention methods but significantly lower than dense attention's $O(n^2)$ and the near-quadratic complexity of LongCoder and SparseCoder. Extensive experiments on LCC and CrossCodeEval benchmarks demonstrate that SabreCoder reduces TTFT by 45-55% while maintaining accuracy within 3% of dense attention. Compared to general sparse methods, SabreCoder improves EM by 47% on LCC and 283% on CrossCodeEval. Compared to code-specific methods like LongCoder, SabreCoder achieves 49% faster inference with comparable accuracy.

Our main contributions are:

- We propose **SabreCoder**, a training-free structure-aware block-sparse attention mechanism that bridges logical and computational sparsity for long-code completion. To the best of our knowledge, this is the first code-aware sparse attention method to achieve genuine hardware acceleration through block-sparse execution.
- We introduce a semantic chunk-based attention framework with chunk-level sparse attention patterns. These patterns effectively model both short-range and long-range code depen-

dencies while naturally mapping to block-sparse computation for genuine speedup.

- We are the first to perform sparse modeling specifically for repository-level code completion scenarios. This covers more realistic code completion workflows and addresses cross-file dependencies that are critical in modern software development.
- We conduct extensive experiments across two representative long-code completion scenarios. Results show that SabreCoder reduces TTFT by 45-55% while maintaining within 3% of dense attention accuracy, significantly outperforming both general and existing code-specific sparse methods. We release the code and data at `https://anonymous.4open.science/r/SabreCoder`.

2 Related Work

2.1 Long Code Completion

Code completion is a fundamental task in software development, where models predict subsequent code given a prefix context. Early work focuses on single-file completion with limited context (Raychev et al., 2014; Li et al., 2017). Recent advances explore two main directions. **Single-file long-code completion** extends context windows to handle longer files. LongCoder (Guo et al., 2023) proposes window-based sparse attention with global imports and fixed-distance bridges. CodeLlama (Roziere et al., 2023) extends context to 100k tokens through continued pre-training. **Repository-level code completion** requires modeling cross-file dependencies. RepoCoder (Zhang et al., 2023a) introduces retrieval-augmented generation for repository context. CrossCodeEval (Ding et al., 2023) provides a multilingual benchmark for this task. RepoFusion (Shrivastava et al., 2023) proposes query-aware retrieval with multi-file context aggregation. RepoHyper (Phan et al., 2025) employs semantic graph traversal for context selection. Despite progress, quadratic attention complexity remains a bottleneck when handling long concatenated contexts.

2.2 Sparse Attention Mechanisms

Sparse attention reduces transformer complexity by attending to token subsets. **General sparse methods** employ fixed patterns without domain knowledge. Sliding window attention (Child, 2019) restricts tokens to local neighborhoods. Longformer (Beltagy et al., 2020) combines local windows with task-specific global tokens. BigBird (Zaheer et al., 2020) adds random attention to local and global patterns. Recent dynamic methods adapt patterns during inference. StreamingLLM (Xiao et al., 2023) identifies attention sinks (initial tokens) critical for performance. MInference (Jiang et al., 2024) constructs sparse patterns by analyzing attention distributions. H2O (Zhang et al., 2023b) evicts low-attention tokens from KV cache. **Code-specific sparse methods** leverage code structure. LongCoder (Guo et al., 2023) treats imports as globally visible and uses periodic bridges for long-range dependencies. SparseCoder (Wang et al., 2024) constructs identifier-based sparse patterns. However, these methods achieve only logical sparsity. Irregular patterns prevent efficient GPU execution, causing limited or negative speedup. SabreCoder addresses this gap through block-sparse mapping that delivers genuine computational acceleration.

3 Method

Figure 1 shows the overall architecture of SabreCoder. Given a code prefix, SabreCoder works in three stages: (1) parsing code into semantic chunks and extracting inter-chunk dependencies, (2) constructing chunk-level sparse patterns through intra-chunk, dependency-based, similarity-based, and global attention, and (3) mapping chunk-level patterns to GPU-friendly block-sparse format with custom Triton kernels for genuine hardware acceleration.

3.1 Code Chunking and Dependency Parsing

3.1.1 Semantic Chunking Strategy

We denote the input code prefix as $X = \{x_1, x_2, ..., x_n\}$, where n is the sequence length. We parse X into m semantic chunks $C = \{c_1, c_2, ..., c_m\}$.

Single-file completion. We use tree-sitter (Brunsfeld, 2018) to parse the code into an abstract syntax tree (AST). We select specific node types as chunks based on code semantics. These node types include function definitions, class definitions, and import statements. Formally, each chunk c_i contains a continuous span of tokens:

$$c_i = \{x_s, x_{s+1}, ..., x_e\} \quad (1)$$

where s and e denote the start and end positions.

Repository-level completion. We follow the RAG paradigm for repository-level tasks. We first retrieve k relevant code snippets from the repository. Each retrieved snippet is directly treated as a chunk. The preceding code is parsed into chunks using the same method as single-file completion. This gives us $m = k + m'$ chunks in total, where m' is the number of chunks from the preceding code.

3.1.2 Dependency Graph Construction

We extract dependencies between chunks to capture code relationships. We use tree-sitter to analyze the AST and identify dependency edges. These dependencies include import relationships, function calls, class inheritance, and variable references.

We construct a directed graph $G = (V, E)$, where $V = C$ represents chunks as nodes. An edge $(c_i, c_j) \in E$ exists if chunk c_i depends on chunk c_j. For example, if c_i calls a function defined in c_j, we add an edge from c_i to c_j.

3.2 Chunk-Level Sparse Pattern Construction

We design four complementary sparse attention patterns. Each pattern captures different types of code relationships. We combine these patterns to form the final sparse attention mask.

3.2.1 Intra-Chunk Attention

Tokens within the same chunk often have strong semantic relationships. We preserve full attention within each chunk. For any two tokens x_i and x_j in the same chunk c_k, we set:

$$M_{\text{intra}}[i, j] = 1 \text{ if } x_i, x_j \in c_k \tag{2}$$

where M_{intra} is the intra-chunk attention mask.

This pattern ensures that related tokens in the same syntactic unit can attend to each other. It preserves local context within functions, classes, and statements.

3.2.2 Dependency-Based Attention

Code dependencies indicate semantic relationships between chunks. We allow chunks to attend to their dependent chunks based on the dependency graph G.

For any two tokens $x_i \in c_p$ and $x_j \in c_q$, we set:

$$M_{\text{dep}}[i, j] = 1 \text{ if } (c_p, c_q) \in E \tag{3}$$

where M_{dep} is the dependency-based attention mask.

This pattern captures explicit code dependencies. For example, a function can attend to the functions it calls. A class can attend to its parent class.

3.2.3 Similarity-Based Attention

Not all code relationships are explicit in the dependency graph. Some chunks are semantically related but lack explicit dependencies. We use embedding similarity to capture these implicit relationships.

For each chunk c_i, we compute its embedding as the mean of token embeddings:

$$\mathbf{e}_i = \frac{1}{|c_i|} \sum_{x_j \in c_i} \mathbf{E}(x_j) \tag{4}$$

where $\mathbf{E}(x_j)$ is the token embedding from the backbone model. We obtain these embeddings through direct lookup. This requires no forward pass and incurs limited overhead.

We compute cosine similarity between chunk embeddings:

$$\text{sim}(c_i, c_j) = \frac{\mathbf{e}_i \cdot \mathbf{e}_j}{||\mathbf{e}_i|| \cdot ||\mathbf{e}_j||} \tag{5}$$

For each chunk c_i, we select the top-α most similar chunks, where α is a percentage threshold. For any two tokens $x_i \in c_p$ and $x_j \in c_q$, we set:

$$M_{\text{sim}}[i, j] = 1 \text{ if } c_q \in \text{TopK}_\alpha(c_p) \tag{6}$$

where $\text{TopK}_\alpha(c_p)$ returns the top-α similar chunks to c_p.

In our experiments, we use the token embedding layer from the backbone model. Since these embeddings are obtained through direct lookup without forward propagation, the computational overhead is negligible.

3.2.4 Global Attention

Some code elements require global visibility. We identify these special tokens based on rules. These tokens include import statements and function signatures. Note that for functions, we only mark the signature part, not the entire function body.

Let $S \subset X$ denote the set of special tokens. For any token $x_i \in S$ and any token $x_j \in X$, we set:

$$M_{\text{global}}[i, j] = M_{\text{global}}[j, i] = 1 \tag{7}$$

This pattern ensures that global information is accessible to all tokens.

3.2.5 Pattern Combination

We combine the four patterns by taking their union. The final chunk-level sparse mask is:

$$M_{\text{chunk}} = M_{\text{intra}} \cup M_{\text{dep}} \cup M_{\text{sim}} \cup M_{\text{global}} \tag{8}$$

This combined mask preserves multiple types of code relationships while maintaining sparsity.

3.3 Block-level Sparse Mapping

3.3.1 Chunk-to-Block Mapping

The chunk-level mask M_{chunk} is irregular and cannot directly leverage GPU block-sparse kernels. We map it to a block-sparse format. We divide the attention matrix into blocks of size $B \times B$. For block (p, q), we define:

$$M_{\text{block}}[p, q] = \max_{\substack{pB \le i < (p+1)B \\ qB \le j < (q+1)B}} M_{\text{chunk}}[i, j] \quad (9)$$

This ensures chunk-level sparse patterns are preserved: whenever tokens from two chunks should attend to each other, all corresponding blocks are activated.

3.3.2 Triton Kernel Implementation

We implement custom block-sparse attention kernels using Triton that skip computation for inactive blocks where $M_{\text{block}}[p, q] = 0$. Following the flash attention algorithm, our kernel only loads and computes key-value blocks marked as active in M_{block}, achieving genuine hardware acceleration by avoiding sparse region computation—unlike logical sparsity methods where irregular patterns prevent efficient execution (detailed implementation in Appendix B).

3.4 Extension to RAG-Based Completion

Repository-level code completion introduces cross-file dependencies that require modeling retrieved snippets alongside the preceding code. SabreCoder naturally extends to this scenario through its chunk-based design. Each retrieved snippet is treated as an independent chunk without further subdivision, respecting semantic boundaries while avoiding parsing overhead. The preceding code is still parsed into fine-grained chunks using tree-sitter, with dependencies extracted only within it.

The four sparse attention patterns handle this asymmetry naturally: intra-chunk attention maintains coherence within each snippet, dependency-based attention connects chunks in the preceding code, similarity-based attention bridges retrieved snippets with preceding code and captures inter-snippet relationships, and global attention ensures visibility of important elements. This extension requires no architectural changes, demonstrating SabreCoder's flexibility across different completion scenarios.

3.5 Complexity Analysis

We analyze the time complexity of SabreCoder. Let n be the sequence length and L be the average chunk size. The intra-chunk attention operates within each chunk, contributing $O(nL)$ complexity. The dependency-based attention connects related chunks with sparse edges. Let k denote the average number of dependencies per chunk. This contributes $O(knL)$ complexity. The similarity-based attention selects top-k similar chunks for each chunk, also contributing $O(knL)$ complexity. The global attention involves a small number of special tokens attending to all positions. Since the number of global tokens is also a small constant, this adds $O(n)$ complexity. Combining these patterns, the overall complexity is approximately $O(knL)$ (dominated by the dependency and similarity terms). In practice, $kL \ll n$, making the complexity close to linear with respect to sequence length. This is significantly lower than dense attention's $O(n^2)$ complexity. Our complexity is comparable to general sparse methods like StreamingLLM and MInference. However, unlike code-specific methods such as LongCoder and SparseCoder that only achieve logical sparsity, our block-sparse format delivers genuine computational speedup through efficient GPU execution.

4 Experimental Setup

4.1 Datasets

We evaluate SabreCoder on two public datasets covering two common long-code completion scenarios:

LCC (Long Single-File Code Completion) (Guo et al., 2023) features single-file completion tasks across three programming languages (Python, Java, C#) with context lengths ranging from 1k to 32k tokens. The dataset contains real-world code files from open-source repositories where the model must complete code given the preceding context.

CrossCodeEval (Repository-Level Code Completion) (Ding et al., 2023) is a repository-level code completion benchmark covering multiple programming languages. We select the Python and Java subsets for evaluation. Unlike single-file completion, repository-level tasks require the model to leverage cross-file context to generate accurate completions. Following the standard retrieval-augmented generation (RAG) paradigm for long-context completion, we employ BM25 (Robertson et al., 2009) as the retrieval method to retrieve the

Table 1: Performance comparison on the LCC dataset. ↓ indicates lower is better, ↑ indicates higher is better. Best and second best results (dense methods excluded from ranking).

Method	Python					Java					C#				
	TTFT↓	PPL↓	EM↑	F1↑	ES↑	TTFT↓	PPL↓	EM↑	F1↑	ES↑	TTFT↓	PPL↓	EM↑	F1↑	ES↑
Dense (Eager)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Dense (Triton)	1686.72	1.19	36.43	50.12	60.45	1619.04	1.11	48.89	61.50	68.75	1648.40	1.10	40.00	59.43	68.94
Sliding Window	712.38	3.70	6.98	21.34	32.57	681.00	3.79	12.22	34.25	43.76	689.36	3.20	10.53	33.65	45.36
StreamingLLM	716.91	1.47	24.81	39.13	50.79	689.12	1.39	36.67	51.78	59.04	698.66	1.39	30.53	50.02	61.16
MInference	**695.73**	1.69	20.93	34.19	44.74	**674.42**	1.64	30.00	44.71	52.65	**683.65**	1.67	28.42	47.89	59.33
BigBird	839.58	1.68	20.93	33.93	44.73	816.84	1.62	26.67	42.32	50.84	819.87	1.60	29.47	48.45	59.75
LongCoder	1841.45	1.55	30.23	44.14	54.52	1765.72	1.44	41.11	54.76	62.07	1831.53	1.39	34.74	56.06	65.83
SparseCoder	1776.16	1.52	28.68	43.43	52.66	1721.47	1.37	40.00	53.82	60.56	1741.44	1.37	31.58	52.14	62.92
SabreCoder (Ours)	**931.19**	**1.21**	**36.43**	**49.41**	**59.18**	**977.29**	**1.14**	**47.78**	**60.70**	**68.40**	**913.14**	**1.15**	**38.95**	**57.81**	**69.07**

most relevant code snippets from the repository and concatenate them with the given preceding context to construct the input prompt.

4.2 Baselines

We compare SabreCoder against three categories of attention mechanisms:

Dense Attention. We evaluate two dense attention implementations: Dense (Eager) and Dense (Triton). Dense (Eager) is PyTorch's native attention implementation with standard matrix multiplication. Dense (Triton) uses a dense Triton kernel optimized for memory access patterns.

General Sparse Methods. We compare against four representative approaches: Sliding Window, StreamingLLM (Xiao et al., 2023), MInference (Jiang et al., 2024), and BigBird (Zaheer et al., 2020). Sliding Window restricts each token to attend only to nearby tokens within a fixed window. StreamingLLM employs local windows while retaining initial tokens as attention sinks. MInference applies dynamic sparse attention based on pre-computed patterns. BigBird combines local windows, random sampling, and global tokens.

Code-Specific Sparse Methods. We evaluate against two state-of-the-art code-specific methods: LongCoder (Guo et al., 2023) and SparseCoder (Wang et al., 2024). LongCoder employs local windows, global tokens, and memory tokens for sparsity motivated by how human programmers write code. SparseCoder uses identifier-aware sparse attention patterns to capture dependencies among code identifiers.

4.3 Evaluation Metrics

Efficiency Metric. We use the Time-to-First-Token (TTFT) as the efficiency metric, as it significantly affects user experience when using code completion assistants.

Quality Metrics. We assess completion quality using three metrics: Perplexity, Exact Match, Edit Similarity, and Token-Level F1 score. Perplexity (PPL) measures the model's uncertainty in predicting the next token, with lower values indicating better language modeling capability. Exact Match (EM) measures the percentage of samples where the generated code exactly matches the ground truth. Edit Similarity (ES) measures the ratio of matching characters using the longest common subsequence. Token-Level F1 score (F1) computes the harmonic mean of precision and recall over token sequences.

4.4 Implementation Details

Inference Setup. We conduct all experiments on a single NVIDIA GeForce RTX 3090 GPU with 24GB memory. We use mixed precision (FP16) for inference to reduce memory consumption and accelerate computation. We adopt greedy decoding to ensure consistency of output results across runs. Additionally, to ensure the stability of the TTFT metric, we perform warm-up before each inference.

Backbone Models. We use DeepSeek-Coder-1.3B (Guo et al., 2024) as the backbone model for most experiments. Additionally, we evaluate on Qwen2.5-Coder-0.5B (Hui et al., 2024), Qwen2.5-Coder-1.5B, Qwen2.5-Coder-3B, and StarCoder2-3B (Lozhkov et al., 2024) to verify the generalization in Section 5.4.

5 Evaluation Results

5.1 Overall Performance

Long Single-File Code Completion. Table 1 shows that SabreCoder achieves substantial speedup (45% TTFT reduction on Python) while preserving accuracy comparable to dense attention. Notably, SabreCoder outperforms general sparse methods by large margins in accuracy (47%

Table 2: Performance comparison on the CrossCodeEval dataset. ↓ indicates lower is better, ↑ indicates higher is better. Best and second best results (dense methods excluded from ranking).

Method	Python					Java				
	TTFT↓	PPL↓	EM↑	F1↑	ES↑	TTFT↓	PPL↓	EM↑	F1↑	ES↑
Dense (Eager)	-	-	-	-	-	-	-	-	-	-
Dense (Triton)	1788.46	1.19	17.00	59.55	68.05	1548.62	1.16	19.75	58.18	66.78
Sliding Window	720.05	6.52	1.25	22.46	43.01	651.26	4.79	2.75	27.64	42.96
StreamingLLM	725.53	1.47	4.50	43.25	58.21	659.37	1.49	8.00	45.02	56.73
MInference	**711.82**	1.72	7.25	46.98	60.91	**645.71**	1.75	9.50	43.34	54.70
BigBird	862.62	1.71	6.75	46.65	60.27	785.62	1.72	9.25	43.50	54.97
LongCoder	1881.59	1.43	10.75	51.16	62.79	1721.90	1.40	11.75	50.75	60.79
SparseCoder	1861.18	1.43	10.50	51.86	64.33	1629.54	1.38	15.00	51.95	61.43
SabreCoder (Ours)	**804.21**	**1.23**	**17.25**	**57.69**	**66.92**	**778.14**	**1.19**	**18.75**	**55.68**	**64.81**

higher EM than StreamingLLM) while maintaining similar efficiency, demonstrating the importance of structure-aware sparsity. Interestingly, code-specific methods LongCoder and SparseCoder exhibit slower inference than dense attention despite their logical sparsity. This counter-intuitive result stems from two factors: (1) their irregular attention patterns prevent efficient GPU kernel execution, forcing fallback to element-wise operations, and (2) the overhead of dependency analysis without corresponding computational benefits. In contrast, SabreCoder's block-sparse format enables genuine hardware acceleration, achieving 49% faster inference than LongCoder with comparable accuracy.

Repository-Level Code Completion. Table 2 validates our repository-level modeling. SabreCoder achieves 55% speedup while maintaining accuracy (17.25% vs 17.00% EM), and dramatically outperforms general sparse methods. This demonstrates that treating retrieved snippets as semantic chunks with similarity-based attention effectively captures cross-file dependencies. Code-specific baselines achieve only 10% EM with slower inference than dense attention, confirming SabreCoder as the first method to bridge structure-aware sparsity with computational efficiency in repository-level completion. A qualitative case study in Appendix C.4 further demonstrates how SabreCoder captures multi-hop dependencies.

5.2 Ablation Study

Table 3 validates the necessity of each component. Removing similarity-based attention causes the most severe degradation (EM: 17.25% → 6.25%), confirming its criticality for capturing implicit cross-file relationships. Removing global attention (EM: 11.75%) and intra-chunk attention (EM: 13.75%) both significantly harm accuracy, showing that imports/signatures require global visibility and tokens within syntactic units need full mutual attention. Finally, removing block-level mapping increases TTFT by 32% (804.21ms → 1065.55ms) while degrading accuracy (EM: 17.25% → 7.75%), as token-level sparsity incurs masking overhead without reducing block computation. Additional ablation results on LCC are in Appendix C.1.

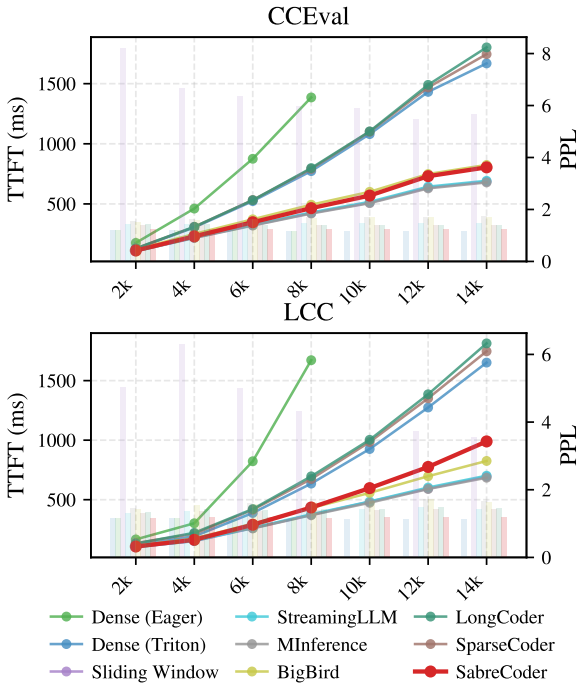

Figure 2: Scaling results on context length for LCC (top) and CCEval (bottom) datasets. The solid lines indicate TTFT (ms) corresponding to the left axis, and the shaded bars indicate PPL corresponding to the right axis. SabreCoder shows significantly better latency scaling while maintaining competitive perplexity.

5.3 Scaling on Context Length

Figure 2 demonstrates the scaling behavior as context length increases from 2k to 14k tokens. Dense (Triton) exhibits near-quadratic growth in TTFT,

Table 3: Ablation study on the CrossCodeEval dataset. ↓ indicates lower is better, ↑ indicates higher is better. Performance degradation compared to the full model is shown in the superscript.

Configuration	Python					Java				
	TTFT↓	PPL↓	EM↑	F1↑	ES↑	TTFT↓	PPL↓	EM↑	F1↑	ES↑
SabreCoder	**804.21**	**1.23**	**17.25**	**57.69**	**66.92**	**778.14**	**1.19**	**18.75**	**55.68**	**64.81**
w/o Intra-chunk Attn.	792.09$^{\downarrow 1.5\%}$	1.25$^{\uparrow 1.6\%}$	13.75$^{\downarrow 3.50}$	55.55$^{\downarrow 2.14}$	65.65$^{\downarrow 1.27}$	836.23$^{\uparrow 7.5\%}$	1.20$^{\uparrow 0.8\%}$	14.00$^{\downarrow 4.75}$	48.54$^{\downarrow 7.14}$	58.68$^{\downarrow 6.13}$
w/o Explicit Deps.	812.76$^{\uparrow 1.1\%}$	1.23$^{\uparrow 0.0\%}$	15.75$^{\downarrow 1.50}$	55.61$^{\downarrow 2.08}$	66.33$^{\downarrow 0.59}$	767.89$^{\downarrow 1.3\%}$	1.20$^{\uparrow 0.8\%}$	13.75$^{\downarrow 5.00}$	48.35$^{\downarrow 7.33}$	59.04$^{\downarrow 5.77}$
w/o Similarity Edges	679.28$^{\downarrow 15\%}$	1.29$^{\uparrow 4.9\%}$	6.25$^{\downarrow 11.0}$	46.71$^{\downarrow 10.9}$	60.63$^{\downarrow 6.29}$	633.64$^{\downarrow 18\%}$	1.26$^{\uparrow 5.9\%}$	9.25$^{\downarrow 9.50}$	43.84$^{\downarrow 11.8}$	56.07$^{\downarrow 8.74}$
w/o Global Visibility	858.60$^{\uparrow 6.8\%}$	1.28$^{\uparrow 4.1\%}$	11.75$^{\downarrow 5.50}$	49.89$^{\downarrow 7.80}$	62.69$^{\downarrow 4.23}$	790.31$^{\uparrow 1.6\%}$	1.23$^{\uparrow 3.4\%}$	13.25$^{\downarrow 5.50}$	47.03$^{\downarrow 8.65}$	57.33$^{\downarrow 7.48}$
w/o Block-level	1065.55$^{\uparrow 32\%}$	1.30$^{\uparrow 5.7\%}$	7.75$^{\downarrow 9.50}$	49.13$^{\downarrow 8.56}$	61.53$^{\downarrow 5.39}$	1264.96$^{\uparrow 62\%}$	1.31$^{\uparrow 10\%}$	6.25$^{\downarrow 12.5}$	42.02$^{\downarrow 13.6}$	54.50$^{\downarrow 10.3}$

Table 4: Results of backbone model generalization (TTFT in ms per sample). Due to GPU memory constraints, we evaluate models on 6k prompt budgets.

Model	Method	LCC	CrossCodeEval
Qwen2.5-Coder-0.5B	Dense	151.35	193.37
	SabreCoder	125.23	144.25
Qwen2.5-Coder-1.5B	Dense	400.97	535.22
	SabreCoder	319.17	380.09
Qwen2.5-Coder-3B	Dense	759.69	989.86
	SabreCoder	613.03	726.21
StarCoder2-3B	Dense	768.33	1034.45
	SabreCoder	576.93	689.06

while SabreCoder shows sub-linear scaling. At 14k tokens, SabreCoder achieves 45% speedup on LCC and 55% on CrossCodeEval. General sparse methods maintain low TTFT but suffer from accuracy degradation, while code-specific methods scale poorly. LongCoder and SparseCoder are even slower than dense attention at 14k tokens. Critically, SabreCoder maintains stable perplexity across all lengths, demonstrating that structure-aware sparsity preserves modeling capability while achieving genuine computational speedup.

5.4 Backbone Model Generalization

Table 4 validates SabreCoder across different backbone models and sizes. The speedup ratios remain consistent: approximately 17-20% TTFT reduction on LCC and 25-27% on CrossCodeEval across all tested models. This consistency across model families (Qwen2.5-Coder, DeepSeek-Coder, StarCoder2) and sizes (0.5B to 3B parameters) indicates that SabreCoder's benefits are model-agnostic and work seamlessly with various transformer implementations without requiring model-specific tuning. Larger models benefit more from sparse attention as quadratic complexity dominates total latency, while the architectural independence demonstrates that our chunk-based patterns capture fundamental code structure rather than model-specific features.

5.5 Pre-computation Overhead Analysis

Table 5: Pre-computation Overhead Analysis (Latency in ms per sample).

Stage	LCC	CrossCodeEval
Pre-computation	216.6	200.0
Prefill (SabreCoder)	940.5	791.2
Overhead Ratio	23.0%	25.3%

Table 5 shows that dependency extraction takes 216.6ms on LCC and 200.0ms on CrossCodeEval, representing 23.0% and 25.3% overhead relative to prefill time respectively. This one-time cost is acceptable given the 45-55% speedup from sparse attention. For interactive coding scenarios, dependency graphs can be cached and reused for repeated queries on the same codebase, effectively amortizing the extraction cost to near-zero. The chunk embedding computation requires only direct lookup from the token embedding layer (typically <10ms), ensuring overall pre-computation overhead remains manageable. Additionally, incremental parsing can update only modified regions in production deployments, further reducing overhead for iterative workflows.

6 Conclusion

We propose SabreCoder, a training-free structure-aware block-sparse attention mechanism that achieves genuine computational speedup for long-code completion. By parsing code into semantic chunks, constructing chunk-level sparse patterns, and mapping them to GPU-friendly block-sparse formats, SabreCoder reduces TTFT by 45-55% while maintaining accuracy within 3% of dense attention. Experiments show significant advantages over general sparse methods (47% EM improvement on LCC, 283% on CrossCodeEval) and code-specific approaches (49% faster than LongCoder), demonstrating that bridging logical and computational sparsity is essential for practical deployment.

Limitations

SabreCoder has several main limitations. First, the dependency extraction overhead (21-24% of prefill time) may be noticeable in single-query scenarios, though caching mitigates this in practice. Second, our approach currently supports Python, Java, and C# through tree-sitter parsers; extending to languages without robust parsers requires additional engineering. Third, while block-sparse attention reduces memory-bound operations, the speedup ratio depends on hardware characteristics—GPUs with higher memory bandwidth may see smaller relative gains. Fourth, due to limited GPU memory (24GB), our experiments are conducted on models up to 3B parameters; evaluating SabreCoder on larger models (7B+) requires more powerful hardware to verify scalability. Future work could explore learned sparse patterns that bypass dependency extraction, multi-language parser development, hardware-aware block size optimization, and comprehensive evaluation on larger model scales.

References

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018a. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51(4):1–37.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018b. Learning to represent programs with graphs. In *International Conference on Learning Representations (ICLR)*.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, and 1 others. 2024. LongBench: A bilingual, multitask benchmark for long context understanding. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.

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

Max Brunsfeld. 2018. Tree-sitter - a parser generator tool and an incremental parsing library. https://github.com/tree-sitter/tree-sitter.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*.

Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. 2024. LongLoRA: Efficient fine-tuning of long-context large language models. In *International Conference on Learning Representations (ICLR)*. Oral.

Rewon Child. 2019. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. 2021. Rethinking attention with performers. In *International Conference on Learning Representations (ICLR)*.

Tri Dao. 2024. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, and 1 others. 2024. DeepSeek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Yangruibo Ding, Jinjun Cao, Jiayi Liu, Yun Ding, Siyuan Zhou, Zhihao Han, Chao Shi, Jie Chen, Zhilei Zhang, Ge Li, and 1 others. 2024a. GraphCoder: Enhancing repository-level code completion via coarse-to-fine retrieval based on code context graph. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Yangruibo Ding, Jinjun Liu, Jiawei Li, Jinjun Cao, Jiayi Liu, Zhihao Han, Chao Shi, Jie Chen, Zhilei Zhang, and Ge Li. 2024b. R2C2-Coder: Enhancing and benchmarking real-world repository-level code completion abilities of code large language models. *arXiv preprint arXiv:2406.01359*.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and 1 others. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36:46701–46723.

Yuxiang Ding, Li Lyna Zhang, Shuo Zhang, Ruoming Xu, Yuqing Shao, Xin Yuan, Dongmei Zhang, and Jianguo Wei. 2024c. LongRoPE: Extending LLM context window beyond 2 million tokens. In *International Conference on Machine Learning (ICML)*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A generative model for code infilling and synthesis. In *International Conference on Learning Representations (ICLR)*.

Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Michael R Lyu, and Irwin King Ng. 2022. Code structure–guided transformer for source code summarization. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–32.

Tao Ge, Jing Hu, Lei Wang, Xun Wang, Si-Qing Chen, and Furu Wei. 2024. In-context autoencoder for context compression in a large language model. In *International Conference on Learning Representations (ICLR)*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, and 1 others. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations (ICLR)*.

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. 2023. Longcoder: A long-range pretrained language model for code completion. In *International Conference on Machine Learning*, pages 12098–12107. PMLR.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. Data movement is all you need: A case study on optimizing transformers. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 3, pages 711–732.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H Abdi, Dongsheng Li, Chin-Yew Lin, and 1 others. 2024. Minference 1.0: Accelerating prefilling for long-context llms via dynamic sparse attention. *Advances in Neural Information Processing Systems*, 37:52481–52515.

Yuexiu Jiang, Xiaohong Su, Christoph Treude, and Tiantian Wang. 2022. Precise learning of source code contextual semantics via hierarchical dependence structure and graph attention networks. *Journal of Systems and Software*, 189:111298.

Hong Jin Kang, Tien H Yee, Shashvat Chandra, and David Lo. 2020. Blended, precise semantic program embeddings. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. 2020. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *International Conference on Machine Learning (ICML)*.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer. In *International Conference on Learning Representations (ICLR)*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, and 1 others. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2017. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573*.

Jia Liu, Yifan Wu, Yizheng Ding, Yitong Luo, Yan Wang, Haoyu Zhao, Jinjun Cao, Chao Shi, Weisong Xu, Ge Li, and Ping Luo. 2024a. ComplexCodeEval: A benchmark for evaluating large code models on more complex code. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Linfeng Liu, Hoan Nguyen Nam Quoc Shi, Jie Zhou, and 1 others. 2024b. AST-T5: Structure-aware pretraining for code generation and understanding. *arXiv preprint arXiv:2401.03003*.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024c. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics (TACL)*, 12:157–173.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2024d. RepoBench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations (ICLR)*.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Huang, Mengdi Jia, Sizhe Yu, Qingzhao Wang, Liangsheng Zhao, Xin Liu, Zhiqiang Zhang, and 1 others. 2024e. CacheGen: KV cache compression and streaming for fast large language model serving. In *ACM SIGCOMM Conference on Applications, Technologies,*

Architectures, and Protocols for Computer Communication.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Maxim Milakov and Natalia Gimelshein. 2018. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*.

Amirkeivan Mohtashami and Martin Jaggi. 2023. Landmark attention: Random-access infinite context length for transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*.

NVIDIA Corporation. 2021. NVIDIA cuSPARSE library documentation. https://docs.nvidia.com/cuda/cusparse/.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. 2024. YaRN: Efficient context window extension of large language models. In *International Conference on Learning Representations (ICLR)*.

Huy N Phan, Hoang N Phan, Tien N Nguyen, and Nghi DQ Bui. 2025. Repohyper: Search-expand-refine on semantic graphs for repository-level code completion. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pages 14–25. IEEE.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 5.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, pages 419–428.

Siyu Ren, Qi Jia, and Kenny Zhu. 2023. Context compression for auto-regressive transformers with sentinel tokens. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Stephen Robertson, Hugo Zaragoza, and 1 others. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Disha Shrivastava, Denis Kocetkov, Harm De Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023. Repofusion: Training code models to understand your repository. *arXiv preprint arXiv:2306.10998*.

Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2020. TreeGen: A tree-based transformer architecture for code generation. In *AAAI Conference on Artificial Intelligence*.

Ze Tang, Xiaoyu Shen, Chuanyi Li, Jidong Zeng, Yuchong Luo, and Chengyang Zhang. 2022. AST-Trans: Code summarization with efficient tree-structured attention. In *International Conference on Software Engineering (ICSE)*.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2023. Efficient transformers: A survey. *ACM Computing Surveys*, 55(6):1–28.

Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*.

Sindhu Tipirneni, Ming Zhu, and Chandan K Reddy. 2024. StructCoder: Structure-aware transformer for code generation. *ACM Transactions on Knowledge Discovery from Data*, 18(6):1–22.

Yang Wang, Chen Zhang, Zhen Xie, Cong Guo, Yunxin Liu, and Jiayu Qiu. 2021. Dual-side sparse tensor core. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

Yanlin Wang, Yanxian Huang, Daya Guo, Hongyu Zhang, and Zibin Zheng. 2024. Sparsecoder: Identifier-aware sparse transformer for file-level code summarization. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 614–625. IEEE.

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 (ICML)*.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*.

Ruyi Xu, Guangxuan Xiao, Haofeng Huang, Junxian Guo, and Song Han. 2025. XAttention: Block sparse attention with antidiagonal scoring. *arXiv preprint arXiv:2501.04489*.

Dongjie Yang, Xiaodong Han, Yan Gao, Yao Hu, Shilin Zhang, and Hai Zhao. 2024a. PyramidInfer: Pyramid KV cache compression for high-throughput LLM inference. In *Findings of the Association for Computational Linguistics (ACL)*.

Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. 2024b. Gated linear attention transformers with hardware-efficient training. In *International Conference on Machine Learning (ICML)*.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and 1 others. 2020. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297.

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

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *International Conference on Software Engineering (ICSE)*.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, and 1 others. 2023b. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36:34661–34710.

Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations (ICLR)*.

A Extended Related Work

This section provides a comprehensive review of related work across five key research areas that inform SabreCoder's design.

A.1 Sparse Attention Mechanisms

Flash Attention series (Dao et al., 2022; Dao, 2024; Shah et al., 2024) introduce IO-aware exact attention using tiling and optimized work partitioning, providing foundational techniques for SabreCoder's efficient kernel implementation. Linear attention methods (Katharopoulos et al., 2020; Choromanski et al., 2021; Yang et al., 2024b) reduce complexity from $O(n^2)$ to $O(n)$ through kernel approximations, while SabreCoder maintains exact attention within selected sparse blocks. Structured sparse attention methods include Sparse Transformers (Child, 2019) with strided patterns, Reformer (Kitaev et al., 2020) with locality-sensitive hashing, and XAttention (Xu et al., 2025) using antidiagonal scoring. Unlike these methods that use fixed patterns or statistical proxies, SabreCoder leverages explicit code structure from AST parsing. KV cache optimization methods (Yang et al., 2024a; Liu et al., 2024e) complement sparse attention by reducing memory costs and could be combined with SabreCoder for further efficiency gains.

A.2 Code Completion and Code LLMs

Recent Code LLMs including StarCoder 2 (Lozhkov et al., 2024), DeepSeek-Coder (Guo et al., 2024; DeepSeek-AI et al., 2024), and InCoder (Fried et al., 2023) establish strong baselines but face efficiency challenges with dense attention on long contexts. Repository-level completion methods (Liu et al., 2024d; Wu et al., 2024; Ding et al., 2024a,b) focus on retrieval strategies, while SabreCoder addresses the complementary challenge of efficiently processing retrieved contexts. Benchmarks including BigCodeBench (Zhuo et al., 2024), ComplexCodeEval (Liu et al., 2024a), and LiveCodeBench (Jain et al., 2024) emphasize real-world complexity where structural understanding is crucial. Structure-aware models (Tipirneni et al., 2024; Sun et al., 2020; Liu et al., 2024b; Tang et al., 2022) demonstrate that incorporating code structure improves performance; SabreCoder extends this by using structure to guide sparse attention patterns.

A.3 Long Context Modeling

Positional encoding extensions (Peng et al., 2024; Ding et al., 2024c; Chen et al., 2023) enable handling longer sequences, making efficient attention mechanisms increasingly important. LongLoRA (Chen et al., 2024) demonstrates that sparse attention during training can approximate full attention at inference, supporting SabreCoder's approach. Landmark Attention (Mohtashami and Jaggi, 2023) uses block-based selection parallel to SabreCoder's structure-aware approach. Context

compression methods (Ge et al., 2024; Ren et al., 2023) represent alternative approaches to handling long contexts. Lost in the Middle (Liu et al., 2024c) discovers U-shaped performance curves, motivating structure-aware attention that guides models to relevant code regardless of position. LongBench (Bai et al., 2024) establishes standard evaluation protocols for long-context understanding.

A.4 GPU Kernel Optimization

Triton (Tillet et al., 2019) provides the compiler infrastructure for SabreCoder's custom block-sparse attention kernels. Block-sparse GPU implementations (NVIDIA Corporation, 2021; Zhu et al., 2019; Wang et al., 2021) establish best practices for implementing sparse operations efficiently on GPUs. Hardware-aware optimization works (Ivanov et al., 2021; Pope et al., 2023) emphasize that achieving genuine speedup requires minimizing data movement and understanding hardware characteristics—principles guiding SabreCoder's implementation.

A.5 Program Analysis and Code Structure

AST-based representations (Zhang et al., 2019; Alon et al., 2019; Zügner et al., 2021) provide foundational evidence that structural code features are essential for neural models. Graph neural networks for code (Allamanis et al., 2018b; Guo et al., 2021) demonstrate how code structure can be incorporated into Transformer attention masks—a direct parallel to SabreCoder. Structure-aware transformers (Gao et al., 2022; Kang et al., 2020) and dependency analysis (Jiang et al., 2022) support SabreCoder's multi-pattern attention design. Foundational surveys (Allamanis et al., 2018a; Tay et al., 2023) contextualize SabreCoder within the broader landscape of efficient transformers and code understanding. Retrieval-Augmented Generation (Lewis et al., 2020) addresses a complementary challenge; combining RAG with sparse attention could enable even more efficient repository-level completion.

B Implementation Details

B.1 Triton Kernel Implementation

We implement SabreCoder using custom Triton kernels for efficient sparse attention computation. The kernel operates on block-level sparsity where each block is 64×64 tokens. Algorithm 1 shows the core computation logic.

Algorithm 1 Block-Sparse Attention Kernel

Require: $Q, K, V \in \mathbb{R}^{B\times H\times L\times D}$, block indices I, block counts C
Ensure: $O \in \mathbb{R}^{B\times H\times L\times D}$

$M_Q \leftarrow \lceil L/64 \rceil$
for $i = 0$ **to** $M_Q - 1$ **do**
  Load query block $Q_i \in \mathbb{R}^{64\times D}$
  Initialize: $acc \leftarrow 0$, $l_i \leftarrow 0$, $m_i \leftarrow -\infty$
  for $j = 0$ **to** $C[i] - 1$ **do**
    $k \leftarrow I[i, j]$ {Get active block index}
    Load key block K_k and value block V_k
    $S \leftarrow Q_i K_k^T / \sqrt{D}$ {Compute attention scores}
    Apply causal mask to S
    Update acc, l_i, m_i via online softmax
  end for
  $O_i \leftarrow acc/l_i$ {Normalize output}
end for

The kernel achieves true sparse acceleration by skipping computation for blocks not in the active set. We use online softmax (Milakov and Gimelshein, 2018) to handle numerical stability without materializing the full attention matrix.

B.2 Code Segmentation with Tree-sitter

We choose tree-sitter for code parsing due to its robust handling of incomplete or partial code. Unlike traditional parsers that fail on syntax errors, tree-sitter employs a GLR (Generalized LR) parsing algorithm with error recovery, allowing it to generate partial Abstract Syntax Trees (ASTs) even when code is syntactically incomplete. When encountering errors, tree-sitter marks problematic nodes as ERROR nodes while continuing to parse surrounding valid code. This error-tolerant design, combined with its incremental parsing capability, makes tree-sitter particularly suitable for processing real-world codebases that may contain temporarily incomplete files or syntax errors during development.

We use tree-sitter parsers to extract code structure across Python, Java, and C#. The segmentation process creates chunks for functions, classes, methods, and imports. For each chunk, we record its byte span, line range, and signature region (definition line plus first docstring line for Python, similar for Java/C#).

We ensure complete line coverage by creating MODULE_CODE chunks for any uncovered regions between semantic chunks. This prevents un-

Table 6: Ablation study on the LCC dataset.

Configuration	Python					Java					C#				
	TTFT	PPL	EM	F1	ES	TTFT	PPL	EM	F1	ES	TTFT	PPL	EM	F1	ES
SabreCoder	931.19	1.21	36.43	49.41	59.18	977.29	1.14	47.78	60.70	68.40	913.14	1.15	38.95	57.81	69.07
w/o Intra-chunk Attn.	936.17	1.23	27.13	42.84	53.44	953.04	1.15	35.56	54.23	62.36	911.88	1.17	32.63	52.83	64.91
w/o Explicit Deps.	925.25	1.22	30.23	46.43	56.09	942.59	1.14	38.89	56.62	64.85	885.23	1.14	36.84	55.86	66.04
w/o Similarity Edges	814.81	1.24	27.13	43.09	52.54	759.86	1.19	31.11	51.00	60.54	730.97	1.16	31.58	52.89	63.83
w/o Global Visibility	927.41	1.36	15.50	30.62	44.37	961.71	1.20	30.00	48.47	58.30	925.42	1.21	30.53	52.87	64.17
w/o Block-level	1384.98	1.25	29.46	44.98	55.84	1435.74	1.19	31.11	49.34	58.29	1507.58	1.14	34.74	55.08	65.12

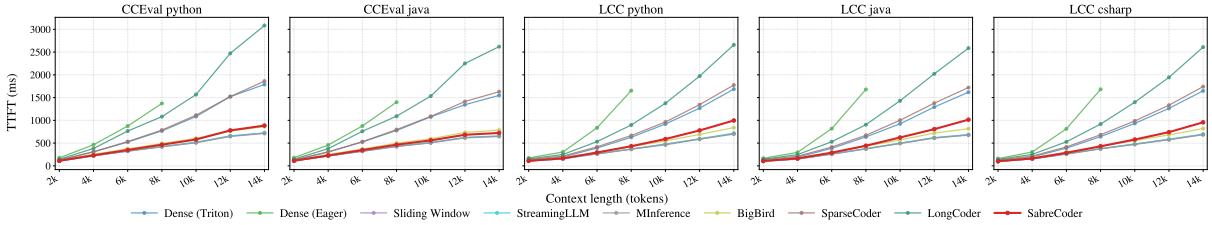

Figure 3: Detailed TTFT (ms) scaling results on five different programming language subsets. SabreCoder consistently maintains superior scaling efficiency compared to both dense and code-specific sparse baselines as the context length increases.

defined behavior in token-to-chunk mapping. The implementation uses an iterative depth-first traversal to avoid recursion limits on deeply nested code.

B.3 Cross-Reference Analysis

We extract function calls and class inheritance using tree-sitter's syntax tree. For function calls, we locate nodes of type `call` (Python), `method_invocation` (Java), or `invocation_expression` (C#), then extract the callee name from the last identifier in the subtree. For inheritance, we extract base type names from class declaration nodes.

The cross-reference analyzer is best-effort and name-based. It does not resolve namespaces or perform full type analysis, making it efficient and robust to incomplete or invalid code in retrieval contexts.

B.4 Chunk Similarity Computation

We compute chunk similarity using token embeddings from the model's embedding layer. For each chunk, we extract up to M_{chunk} tokens (default 256), embed them, and take the mean embedding as the chunk vector. We then compute cosine similarity between all chunk pairs and select the top-k neighbors for each source chunk, where $k = \min(k_{max}, \lfloor p \cdot N_{target} \rfloor)$. Here p is the similarity ratio (default 0.4) and k_{max} is the maximum neighbors per chunk (default 16).

This approach scales to long contexts because we operate on chunk-level vectors rather than full token sequences. We cache chunk vectors across samples with the same code to avoid redundant embedding lookups.

C Additional Experimental Results

C.1 Ablation Study on LCC

We provide additional ablation study results on the LCC dataset to complement the analysis presented in Section 5.2. Table 6 shows the performance of SabreCoder and its variants across Python, Java, and C# programming languages. The results are consistent with our findings on CrossCodeEval: removing similarity-based attention causes significant accuracy drops (e.g., EM decreases from 36.43% to 27.13% on Python), while removing block-level mapping increases TTFT by approximately 49% (from 931.19ms to 1384.98ms on Python), confirming that both semantic modeling and efficient execution are essential for achieving the balance between speed and accuracy.

C.2 Hyperparameter Tuning Process

We tune chunk similarity hyperparameters using a two-stage grid search. In stage one, we fix $k_{max} = 8$ and search over top-$p \in \{0.2, 0.3, 0.4, 0.5\}$ on a 100-sample validation set. In stage two, we fix the best p from stage one and search over $k_{max} \in \{8, 16, 32\}$. The crossfile ratio is set to half of the main ratio by default, then fine-tuned independently for CCEval datasets. This process balances computational cost (grid search) with per-

Figure 4: A case from the CrossCodeEval dataset.

formance (per-dataset tuning). We find that the optimal parameters vary significantly across languages: C# benefits from higher similarity ratios likely due to more verbose code, while Python works best with moderate ratios.

C.3 Detailed Latency Scaling across Subsets

Figure 3 provides a comprehensive breakdown of the TTFT scaling trends across five different programming language subsets. Consistent with the average trends observed in the main text, SabreCoder (red line) exhibits stable, near-linear growth in TTFT across all scenarios, significantly outperforming the Dense (Eager) baseline which suffers from OOM at 8k tokens. Notably, while some code-specific sparse models are designed for long contexts, their computational overhead in managing complex sparse patterns causes their TTFT to exceed even the Dense (Triton) implementation at 14k tokens in several cases. In contrast, SabreCoder effectively maintains the TTFT below 1000ms in all tested subsets, demonstrating the robustness and generalizability of our structure-aware sparsity across diverse coding syntaxes and sequence lengths.

C.4 Case Study: Long-Range Multi-Hop Dependency Capture

Figure 4 illustrates SabreCoder's ability to capture long-range multi-hop dependencies. In this repository-level completion example, the task requires calling `Utils.agg_diff_fe_calc(self.detect_data, self.anomaly_duration)`, which involves understanding: (1) the `Utils` class imported from `common.utils`, (2) the correct method name based on the class's data processing purpose, and (3) the parameter `self.anomaly_duration` from the constructor rather than the constant `RATE`. General sparse methods fail with incorrect predictions like `aggregate_list` or `get_agg_list`, while code-specific methods LongCoder and SparseCoder predict the correct method but use the wrong parameter `RATE`. SabreCoder successfully captures this multi-hop dependency chain through dependency-based attention (connecting imports and class structure), intra-chunk attention (preserving constructor-to-usage relationships), and similarity-based attention (identifying related data processing operations), demonstrating the effectiveness of structure-aware sparsity for complex repository-level reasoning.