SYNKV: RETHINKING ATTENTION STRUCTURE IN LLMS FOR LONG-CONTEXT CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

As LLM-based code generation scales to repository-level inference, context lengths routinely exceed 100K tokens, sharply intensifying GPU memory pressure. Existing KV-cache compression methods remain limited in this setting: token-level selection struggles under aggressive KV budgets, while fixed-size grouping can break coherent code structures and substantially degrade generation performance. Program syntax provides a structural perspective, organizing code into locally coherent blocks. This raises a natural question: **does code-LLM attention mirror this block organization for KV retention?** Query-key interactions reveal syntax-aligned blocks whose recurring properties provide actionable selection signals. Motivated by this finding, we develop SynKV, a syntax-aware block-level framework for KV retention in long-context code generation that alleviates inference memory pressure while preserving generation quality. SynKV restructures flat token sequences into syntax-aligned blocks, then combines hierarchical dependency retrieval with lightweight summary-based relevance estimation to retain critical KV entries. Across four diverse code-generation benchmarks and six complementary metrics, SynKV advances the frontier in both memory scalability and generation quality. Notably, SynKV exhibits pronounced advantages under extreme long-context inference: (i) at the 1M-token scale, SynKV cuts the device-memory footprint from TB scale to tens of GB with only 0.005% KV retention, while existing systems run out of memory; and (ii) SynKV consistently sustains generation quality under aggressive compression, outperforming the best-performing method by over 40%. Beyond SynKV, our findings open a new direction in structured-attention design for scaling long-context LLM inference, balancing memory efficiency and generation quality.

1 INTRODUCTION

Code generation has gained renewed momentum with large language models (LLMs) due to their sophisticated program modeling capabilities, reshaping modern software development with substantial productivity gains (Zan et al., 2023; Jiang et al., 2026; Li et al., 2025). However, as LLM-based code generation evolves from isolated snippets to repository-level reasoning with extended contexts from documentation and execution traces (Liu et al., 2025; Chen et al., 2025; Ashik et al., 2026), code contexts have grown far beyond those of typical natural language tasks, turning the long-standing GPU memory limitation into a critical bottleneck (Ouyang et al., 2025; Gong et al., 2026; Wang et al., 2026). For instance, processing a 128K-token context in RepoBench (Liu et al., 2024) requires over 120 GB of device memory, exceeding the capacity of mainstream single GPUs.

At the heart of long-context memory pressure lies attention processing, where KV cache can occupy up to 70% of inference memory and full-context attention incurs quadratic costs. Meanwhile, attention sparsity provides an opportunity for memory-efficient inference, as substantial low-weight query-key interactions contribute marginally to the final attention output due to softmax's exponential amplification of score differences. Existing studies therefore investigate selective retention of critical KV entries to reduce memory overhead while preserving generation quality (Feng et al., 2026b; Mao et al., 2026). However, most methods rely on token-level retention, as illustrated in Figure 1a, where individually scoring KV entries incurs substantial estimation overhead and makes aggressive compression difficult when critical information is scattered across tokens (Li et al., 2024; Xiao et al., 2024). Grouping tokens can alleviate token-level estimation, but existing methods typ-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

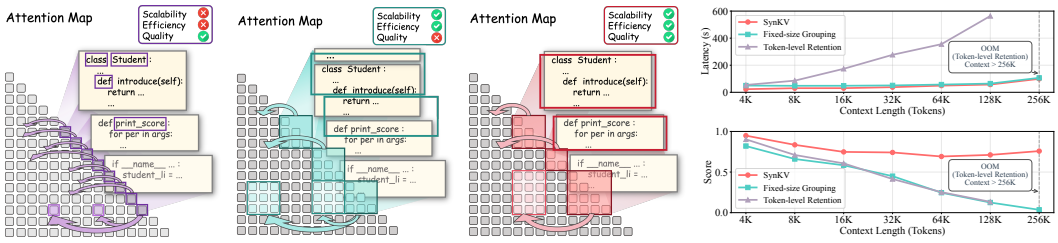
ically rely on fixed-size chunks, as shown in Figure 1b, whose structure-agnostic boundaries may split syntax scopes and break coherent code units, thereby degrading generation (Kim et al., 2026).

Such limitations bring program syntax to the forefront, organizing code into locally coherent blocks. A natural hypothesis follows: **code LLM attention should mirror this block organization KV retention.** Probing this hypothesis, we trace query-key attention patterns back to their enclosing syntactic blocks across representative code-generation models, uncovering a striking consistency: critical interactions concentrate within *syntax-aligned blocks*.

The challenge now shifts to which blocks matter to the current query: exhaustively scoring them all would largely undermine the efficiency gained from block-level retention. Digging deeper into these block-level patterns reveals two recurring properties that provide an actionable signal for KV selection. *Inter-block consistency* shows that queries from the same block tend to share similar attention distributions over other blocks, collapsing per-token relevance estimation into a single block-level decision. *Block-prefix representativeness* presents that prefix tokens within a block tend to receive stronger attention from subsequent queries, making them compact proxies for estimation.

We pursue this insight by developing ***SynKV***, a *syntax-aware block-level* framework for KV retention in long-context code generation. SynKV organizes attention interactions around syntax-aligned blocks and retains critical KV entries at block granularity rather than selecting them individually. This shift consolidates scattered critical information through an expanded retention scope anchored to syntax boundaries, sustaining effective compression beyond 256K tokens with 46.3% higher generation quality than grouping-based methods.

Based on block-level abstraction, SynKV further tackles the challenge of efficient and accurate critical-block identification for long-context KV retention. Code relevance is rarely governed by distance alone: useful blocks may be far apart yet tightly connected through scopes, calls, or cross-file dependencies. SynKV therefore combines block hierarchy with structural relations to recover non-local block relevance beyond linear locality. Long contexts also make exhaustive block scoring impractical. SynKV sidesteps this cost by retaining representative prefix tokens as lightweight summaries to estimate block relevance without loading complete blocks.



(a) Token-level Retention (b) Fixed-size Grouping (c) SynKV (Ours) (d) Latency-Quality Trade-off

Figure 1: Comparison of KV-retention strategies.

We evaluate SynKV comprehensively on four diverse code generation benchmarks, covering six performance metrics over approximately 30K samples with context lengths ranging from 1K to 1M tokens. Compared with six state-of-the-art (SOTA) methods spanning KV-cache compression and code-aware prompt compression, SynKV simultaneously pushes the SOTA in memory scalability and quality robustness. Remarkably, the gap widens sharply as inference moves into extreme context lengths. At 1M tokens, SynKV brings device-memory demand down from the TB scale to tens of GB while retaining just 0.005% of KV, while competing systems run out of memory. SynKV also preserves strong generation quality across compression regimes, outperforming the strongest methods by over 40% under aggressive budget. More broadly, our findings point to structured attention as a new direction for scaling long-context LLM inference with limited quality degradation.

2 Related Work

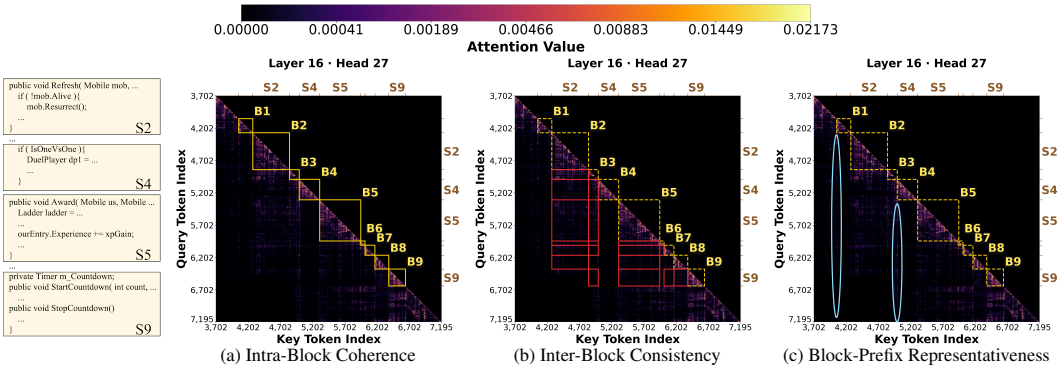
KV Attention patterns. Early studies by Xiao et al. (2024) uncovered static attention patterns centered around initial and recent tokens for efficient long-context generation. Tang et al. (2024) further revealed query-dependent token importance, motivating dynamic importance estimation based on

historical context. Extending this insight, Jiang et al. (2024) identified more diverse attention patterns, including vertical and slash patterns, where certain token positions receive persistent attention or follow recurring attention patterns. Beyond sequence patterns, subsequent studies explored attention heterogeneity across layers (Hao et al., 2025), heads (Fu et al., 2025; Gao et al., 2026), and channels (Xu et al., 2025), revealing uneven importance distributions among model components. Meanwhile, some approaches leverage external prompt-level priors, such as punctuation or token frequency (Ge et al., 2024), to facilitate critical token identification (Ahn et al., 2026; Ramachandran et al., 2026). For code generation, Chen et al. (2026) explored code-specific importance, revealing that structural tokens can be preserved during compression. **Beyond individual token importance, our work uncovers block-level attention regularities in code generation, where critical query-key interactions align with syntax structure, enabling efficient relevance estimation.**

Selective KV Retention. Recent studies translate attention patterns characterized by locality and continuity in natural-language contexts into KV-retention decisions. Zhang et al. (2023) observed attention continuity among nearby historical queries, motivating methods such as SnapKV (Li et al., 2024) and OmniKV (Hao et al., 2025) to leverage recent queries as probes for estimating critical KV entries. Wang et al. (2025) further reduced selection overhead by reusing prefilling attention statistics during decoding. Meanwhile, Jiang et al. (2024) revealed that representative tokens within continuous regions can approximate full attention computation, inspiring subsequent studies to group tokens with fixed sizes for efficient KV selection (Yang et al., 2025; Kim et al., 2025; Xiong et al., 2025). However, such locality-based strategies are less effective for long-context code generation, where critical dependencies are often determined by program structures rather than sequential proximity. **In contrast, our work introduces syntax-aligned block-level KV selection by exploiting hierarchical code structures to identify relevant blocks beyond sequential locality.**

3 MOTIVATION

To probe whether code-LLM attention mirrors program syntax for effective KV retention, we map query–key interactions onto syntax-aligned blocks across representative code-generation models. Beyond previously reported token-level patterns, we uncover a code-specific organization, *intra-block coherence*, where critical interactions exhibit strong within-block concentration. Digging deeper reveals two further regularities, *inter-block attention consistency* and *block-prefix representativeness*, providing a structural basis for efficient critical-block identification, as shown in Figure 2.



(a) Intra-Block Coherence (b) Inter-Block Consistency (c) Block-Prefix Representativeness

Figure 2: Syntax-aligned block-level attention patterns in a *C#* example.

Intra-block coherence. The diagonal regions of attention maps exhibit concentrated attention patterns within the boundaries of syntax-aligned blocks, forming triangular structures around recent code regions. While diagonal attention is commonly associated with local token interactions, this pattern reveals a syntax-aware locality beyond predefined recent-window assumptions, where queries concentrate attention within the same syntax-aligned block. This phenomenon can be attributed to the local semantic closure induced by program syntax. In code generation, syntax-aligned blocks form structurally coherent units, where tokens are strongly related through shared program contexts. Since query-key attention weights reflect learned dependencies among code tokens, tokens within the same syntax-aligned block tend to exhibit stronger aggregate attention relevance (a struc-

tural explanation in Appendix A.1). This concentration suggests syntax-aligned blocks as natural units for block-level KV management, enabling efficient preservation of critical code context.

Inter-block consistency. Beyond the diagonal attention patterns, attention maps exhibit rectangular regions, where the vertical dimension spans queries within a syntax-aligned block and the horizontal dimension spans keys in another block. This pattern indicates that different queries from the same syntax-aligned block tend to exhibit highly consistent attention distributions over external blocks. The consistent attention distributions arises from the shared program context within a syntax-aligned block. Building on intra-block coherence, queries in the same block are dominated by a common block-level representation with only limited token-specific variations, which can be expressed as:

$$q_t = \bar{q}_{B_i} + \epsilon_t, \qquad x_t \in B_i, \quad \|\epsilon_t\| \ll \|\bar{q}_{B_i}\| \tag{1}$$

where $\bar{q}_{B_i}$ denotes the shared representation and ϵ_t captures token-specific variations. Consequently, the attention relevance from queries in B_i to another block B_j remains highly consistent, i.e.,

$$|M(q_t, B_j) - M(q_s, B_j)| \leq C_{ij}\|\epsilon_t - \epsilon_s\| \approx 0 \tag{2}$$

where C_{ij} denotes the sensitivity of block-level attention to query perturbations. This consistency suggests that inter-block relevance can be estimated once per query block rather than independently for every query token, providing the foundation for efficient block-level KV selection.

Block-prefix representativeness. Attention maps exhibit another recurring pattern in which attention regions originate near the prefix of a syntax-aligned block and extend vertically across subsequent query blocks, nearly spanning the entire map. This pattern indicates that block-prefix tokens tend to receive sustained attention from later queries. Such prominence arises because block prefixes often introduce high-level structural and semantic cues, such as declarations, control conditions, and invocation targets, that characterize the role of the block in subsequent code generation. Consequently, block-prefix tokens provide a compact proxy for inter-block attention relevance, substantially reducing the cost of block-level relevance estimation.

4 Design of SynKV

4.1 Overview

SynKV exploits syntax-aligned attention structure in code to organize KV retention around syntax-aligned blocks. The *Syntax-Aligned Block Abstraction* establishes the retention granularity, while *Structural Candidate Retrieval* and *Critical Block Selection* select relevant blocks for candidate-scope formation and final retention determination, respectively. Specifically, SynKV performs block-to-block relevance estimation during prefilling and token-to-block estimation during decoding, with each selected critical-KV set reused throughout the current syntax block. Within this abstraction, the retrieval module narrows the search space guided by hierarchical code relations, forming a candidate pool for the selection module. The latter further estimates their relevance from lightweight block summaries across representative layers and heads, as shown in Figure 3.

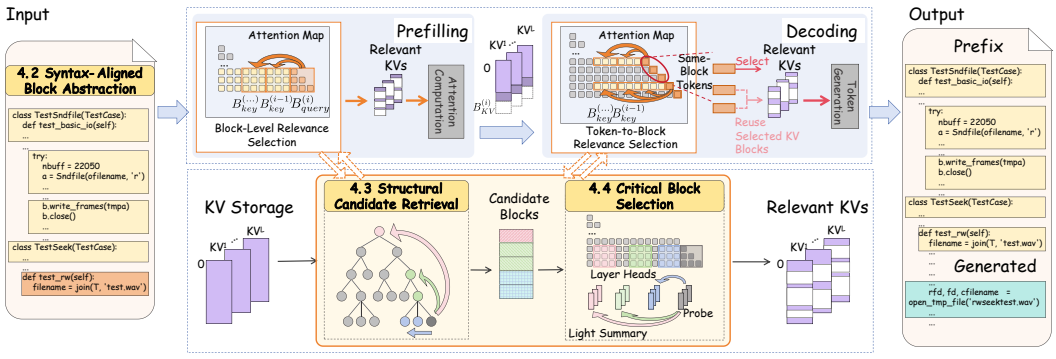


Figure 3: Overview of SynKV's syntax-aligned block-level KV retention framework.

4.2 SYNTAX-ALIGNED BLOCK ABSTRACTION

Syntax-Aligned Block Abstraction groups prefilling contexts and generated code into syntax-defined blocks, each capturing tokens that share local structure. Attention exhibits the same coherence, as queries concentrate on keys within the block, motivating block-level KV selection.

Across diverse language families, block boundaries follow the structural conventions of each syntax: paired tags in XML and matched delimiters in C and Java expose explicit boundaries, whereas indentation-sensitive languages such as Python encode scope through layout. For incomplete contexts, closing boundaries may be absent even when opening boundaries already reveal the active scope hierarchy. An ordered scope stack keeps these open-ended blocks structurally anchored.

We bound syntax-aligned blocks to 128$\sim$1K tokens, balancing fine-grained structural alignment with sufficient context for reliable relevance estimation (Corallo & Papotti, 2024). Oversized blocks are recursively divided along internal syntax boundaries, preventing individual blocks from dominating the KV budget. Undersized blocks are coalesced with adjacent siblings under the same parent scope until reaching the minimum size. The merge never crosses scope boundaries, preserving the original syntax units while expanding only their retention scope.

4.3 STRUCTURAL CANDIDATE RETRIEVAL

Built on the syntax-aligned blocks, Structural Candidate Retrieval narrows the search space before attention-based relevance estimation. Rather than relying on locality-based selection, SynKV restores non-local dependencies from program structure, where distant code blocks may remain tightly related through shared scopes, hierarchical containment, or reference-definition relations. To expose these relations, syntax-aligned blocks are organized into a code hierarchy tree that guides retrieval.

The hierarchy follows program scopes, with each node corresponding to a syntax-aligned block and tree depth reflecting the level of lexical nesting. It captures three structural relations: parent-child containment across nested scopes, sibling relations among same-scope peers, and module-level organization at the root. A lightweight symbol table accompanies each scope to record definitions and imported symbols, allowing references to be traced back to their defining blocks.

For a query block B_q, the candidate set combines three complementary structural sources:

$$\mathcal{C}(B_q) = \mathcal{C}_{\text{recent}}(B_q) \cup \mathcal{C}_{\text{anc}}(B_q) \cup \mathcal{C}_{\text{ref}}(B_q), \tag{3}$$

where $\mathcal{C}_{\text{recent}}$ retrieves preceding sibling blocks for local contextual continuity, $\mathcal{C}_{\text{anc}}$ preserve enclosing scope information, and $\mathcal{C}_{\text{ref}}$ retrieves definition blocks of symbols referenced by B_q.

4.4 CRITICAL BLOCK SELECTION

Critical Block Selection estimates block-level attention dependencies over the retrieved candidates to isolate query-relevant blocks. Leveraging observed inter-block attention consistency, it distills fine-grained token dependencies into a compact set of critical blocks.

Relevance estimation is tailoredto the execution characteristics of each phase: block-level in prefilling and token-to-block in decoding. During prefilling, Critical Block Selection exploits complete query blocks to directly score retrieved candidates at block level. Decoding, in contrast, reveals code one token at a time before the surrounding block is complete. The module therefore ties selection updates to syntax-block transitions, refreshing the selected block set only when the current token maps to a different syntax block from its predecessor; otherwise, it reuses the ongoing selection.

Rather than scoring every token within a block, Critical Block Selection approximates full-block attention with representative tokens, sharply reducing estimation overhead. Guided by block-prefix representativeness in attention patterns, query-block tokens serve as probes against lightweight summaries extracted from candidate prefixes. Each summary retains informative tokens, such as identifiers, keywords, and operators, while filtering formatting content. Import blocks are kept intact, as their short length and explicit symbol-level dependencies make summarization unnecessary.

Block relevance is estimated across representative layers and heads to capture attention dependencies from multiple network perspectives. Exploiting the similarity of neighboring layers and heads, the module samples them uniformly at fixed intervals, retaining broad network coverage without

redundant estimation. For each sampled layer (l) and head (h), the relevance between the query block (B_q) and a candidate block (B_i) is computed as:

$$M_{l,h}(B_q, B_i) = \frac{1}{|Q_p|} \sum_{q \in Q_p(B_q)} \sum_{k \in S(B_i)} A_{l,h}(q, k), \tag{4}$$

where $A_{l,h}(q, k)$ denotes the attention weight between query q and representative token k. The module then aggregated across the sampled layers and heads to characterize each candidate block, retaining the smallest subset that achieves the required attention coverage ratio:

$$C_{\text{rel}}(B_q) = \arg \min_{\mathcal{C} \subseteq C(B_q)} |\mathcal{C}| \quad \text{s.t.} \quad \sum_{B_i \in \mathcal{C}} \text{Score}(B_i) \geq \eta \sum_{B_i \in C(B_q)} \text{Score}(B_i). \tag{5}$$

where $C(B_q)$ denotes the retrieved candidate set and η represents the required attention coverage ratio. Blocks showing outlier attention relevance in individual sampled layers or heads are additionally preserved to avoid overlooking sparse but important dependencies.

5 Evaluation

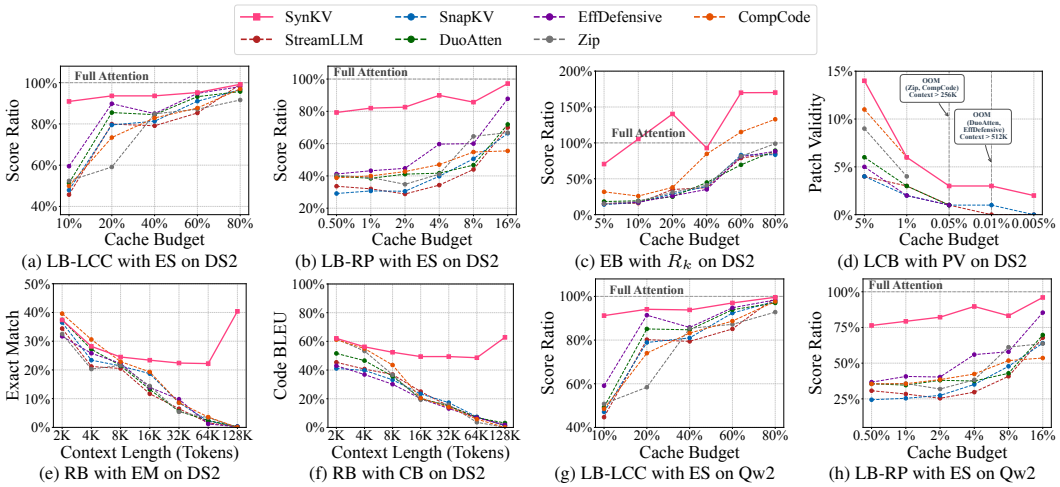
5.1 Setup

Benchmarks. We select four representative LLM-based code generation benchmarks: *LongBench* (*LB*) (Bai et al., 2024), *EvoBench* (*EB*) (Yu et al., 2025), *LongCodeBench* (*LCB*) (Rando et al., 2025), and *RepoBench* (*RB*) (Liu et al., 2024), which broaden existing evaluations by spanning context lengths from 4K to 1M tokens and covering diverse code generation scenarios rather than a single task setting. Specifically, the evaluated workloads ranges from single-file completion with short contexts (hundreds to a few thousand tokens) in LB-LCC, to block-level and cross-file completion with moderate contexts (2K–8K tokens) in LB-RP and EB, and further to repository-level completion and issue resolution with ultra-long contexts (16K–1M tokens) in RB and LCB. These benchmarks comprise about 30K generation samples in total, where all inputs are converted into the fill-in-the-middle format for consistent LLM evaluation.

Metrics. We evaluate code generation across five complementary aspects with six metrics, capturing surface-level similarity, semantic correctness, syntactic validity, dependency awareness, and functional correctness. Given the diverse generation scenarios and objectives covered by these benchmarks, we follow their official evaluation metrics to ensure consistent comparisons. Specifically, surface-level similarity is measured by *Exact Match (EM)* and *Edit Similarity (ES)*, where EM requires exact code matching and ES tolerates minor variations. Semantic correctness is evaluated by *CodeBLEU (CB)* on RepoBench, incorporating syntactic and data-flow information. Syntactic validity is measured by *Patch Validity (PV)* in LongCodeBench. Dependency awareness is evaluated by *Recall@k* (R_k) in EvoBench, measuring whether the generated candidates cover required cross-file dependencies. Functional correctness is evaluated by *Pass@k* (P_k) in EvoBench, which measures the probability that at least one of the k candidates passes all test cases.

Models. We select two representative open-source code generation LLMs for evaluation: DeepSeek-Coder-V2-Lite (DS2) (DeepSeek-AI et al., 2024) and Qwen2.5-Coder-7B (Qw2) (Hui et al., 2024). DS2 is a modern long-context code generation model with a GQA-based architecture (Ainslie et al., 2023), supporting up to 128K tokens. Qw2 is a widely adopted code generation model in recent KV-cache compression studies, serving as a representative evaluation backbone.

Compared Methods. We evaluate SynKV against six representative methods spanning mainstream KV-cache compression and code-aware prompt compression. CodeComp (Chen et al., 2026) incorporates static program analysis into KV-cache optimization for code generation. For general KV-cache compression, we include classical approaches such as StreamingLLM (Xiao et al., 2024), which adopts a sliding-window strategy, and SnapKV (Li et al., 2024), which aggregates attention scores over fixed-size chunks to select KV entries. We further compare with recent SOTA general KV-cache compression methods: DuoAttention (Xiao et al., 2025) and DefensiveKV (Feng et al., 2026a). DuoAttention separates retrieval heads from streaming heads for selective KV retention, while DefensiveKV adopts risk-aware eviction to improve robustness against uncertain observations. Beyond KV-cache compression, we include Zip (Shi et al., 2025), which removes less important tokens based on conditional perplexity. We adapt DuoAttention to the DS2's KV layout through tensor reshaping due to architecture mismatch.



(a) LB-LCC with ES on DS2 (b) LB-RP with ES on DS2 (c) EB with R_k on DS2 (d) LCB with PV on DS2

(e) RB with EM on DS2 (f) RB with CB on DS2 (g) LB-LCC with ES on Qw2 (h) LB-RP with ES on Qw2

Figure 4: Representative performance under varying KV retention ratios across different code generation metrics, benchmarks, and models. Results are reported as ratios to full attention performance.

Implementation Details. We comprehensively evaluate SynKV across four code-generation benchmarks and six complementary performance metrics, covering approximately 30K samples. Spanning context lengths from 1K to 1M tokens, our evaluation considers three KV-retention regimes. Following common settings in prior work, typical compression retains 20%–80% of the KV cache, while aggressive compression reduces retention to 1%–20%. For ultra-long contexts, we further stress SynKV under extreme compression with only 0.005%–1% KV retention. All experiments are conducted on NVIDIA A800 GPUs using Python 3.10, PyTorch 2.4, and CUDA 12.4.

5.2 Performance Preservation

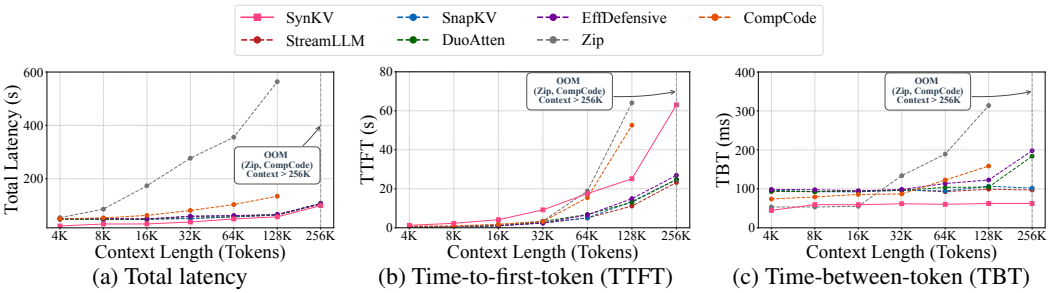
We evaluate SynKV's performance preservation under varying compression ratios compared with full attention, with a particular focus on long-context scenarios. Given the diverse context scales across datasets, we adopt benchmark-specific compression ratio ranges, while fixing compression ratios over different context lengths for LCB and RB with ultra-long contexts.

Overall Analysis. Representative results in Figure 4 (with full details provided in Appendix A.2) that SynKV sustains generation quality as KV budgets shrink across diverse benchmarks and models, extending efficient inference into ultra-long contexts. Under typical compression settings (40%–80% KV retention), SynKV improves the average score by about 4% over the best-performing SOTA methods. For instance, with 60% KV retention on LB-LCC (Figure 4a), SynKV incurs only a 4.8% performance loss, compared with 5.2% for the best-performing method, DefensiveKV (Feng et al., 2026a), benefiting from fine-grained syntax-aligned block modeling for accurate relevant block identification. Notably, SynKV remains robust under aggressive compression by exploiting inter-block consistency, which carries stable block relevance across queries within the same syntax block. With only 1%–10% KV retention on LB-RP and 0.1%–3% on RB (Figures 4b and 4f), SynKV still achieves 63.1% ES and 54.4% CB, while existing methods suffer substantially larger degradation as the cache budget decreases. Even under extreme compression ratios (0.005%–0.01% KV retention), SynKV preserves 3%–6% patch validity by capturing globally relevant blocks while cutting the device-memory footprint from the terabyte scale to tens of gigabytes. The resulting memory headroom extends feasible inference to ultra-long contexts where many compared methods already fail: Zip (Shi et al., 2025) and CodeComp (Chen et al., 2026) encounter out-of-memory (OOM) errors at 256K tokens during query-conditioned perplexity evaluation, while DuoAttention (Xiao et al., 2025) and DefensiveKV (Feng et al., 2026a) fail during 512K-token prefilling.

Metric Analysis. Figure 6 summarizes performance across five code generation evaluation aspects. SynKV consistently outperforms existing methods on all five aspects, with similar trends across models (Appendix A.3). In particular, SynKV shows prominent advantages in semantic correctness and dependency awareness, achieving over 50% higher scores than the best methods. Notably,

SynKV even surpasses full attention in dependency awareness, indicating that syntax-aware block selection better captures dependency-related code regions by allocating attention to structurally relevant contexts. In contrast, the improvement in surface-level similarity is relatively limited, especially for typical contexts, due to repetitive code patterns that reduce the metric sensitivity. Similarly, functional correctness shows smaller gains, largely limited by the base model capability.

Compression Insights. Further analysis of Table 1 (with additional results in Appendix A.4) reveals several insights beyond the performance comparison. First, performance degradation exhibits a threshold behavior as compression increases. As shown in Figure 4c, SynKV maintains stable generation quality under moderate compression (60–80% KV retention), while performance drops sharply when retention decreases below 40%. This indicates that code generation can tolerate the removal of a portion of contextual information as long as critical dependencies are preserved. Therefore, effective KV compression should move beyond fixed-ratio removal and instead prioritize critical context. Second, fixed-size chunk compression may introduce structure-agnostic KV selection, degrading code generation quality. For example, SnapKV (Li et al., 2024) adopts 128-token chunk-based compression but performs worse than even the simple StreamingLLM (Xiao et al., 2024) strategy in many code generation settings. Different from natural language, code generation relies on syntax structures, where arbitrary partitioning across semantic boundaries can disrupt program semantics. Finally, edit similarity alone may not fully reflect code generation quality. Although DefensiveKV (Feng et al., 2026a) achieve relatively high similarity scores, they do not consistently yield comparable improvements in CodeBLEU or pass rates. This suggests that surface-level pattern preservation alone may overlook functional dependencies required for executable correctness.



(a) Total latency (b) Time-to-first-token (TTFT) (c) Time-between-token (TBT)

Figure 5: Generation latency comparison under different context lengths.

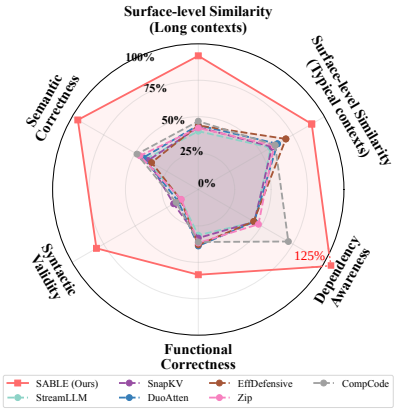


Figure 6: Performance across six metrics on DS2.

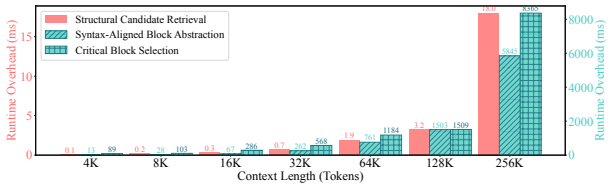


Figure 7: SynKV runtime breakdown.

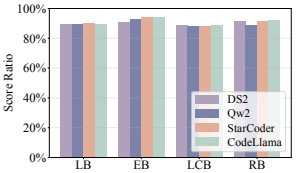


Figure 8: Cross-model generality.

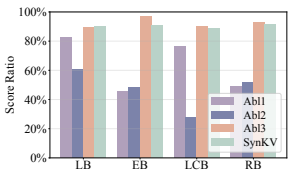


Figure 9: Component ablation.

5.3 Efficiency Tests

We evaluate SynKV's inference efficiency across context lengths of 4K–256K tokens. Figure 5 reports total latency, time-to-first-token (TTFT), and time-between-token (TBT), where TTFT and TBT reflect prefilling and decoding costs, respectively (other results in Appendix A.5). Overall, SynKV reduces latency by 10–52% compared with the best-performing methods. Although SynKV

Table 1: Detailed performance comparison under different KV retention ratios (%).

Method	LB-LCC ES	LB-RP ES	EVO R_1	EVO R_3	EVO R_5	EVO P_1	EVO P_3	EVO P_5	LCB ES	LCB PV	RB EM	RB ES	RB CB
DeepSeek-Coder-V2-Lite (DS2), 20% Cache Budget (2% for LB-RP)									128K tokens for LCB and 64K tokens for RB				
Full Cache	76.59	73.22	4.16	6.27	8.13	13.82	17.09	23.27	OOM	OOM	35.46	70.54	57.23
StreamingLLM (Xiao et al., 2024)	61.19	21.08	1.53	2.42	2.44	1.09	1.82	2.18	2.37	3	1.60	22.52	7.25
SnapKV (Li et al., 2024)	60.86	22.31	1.00	2.19	2.27	1.09	2.18	2.55	3.49	2	2.20	18.26	7.43
DuoAttention (Xiao et al., 2025)	65.45	30.11	0.99	1.59	2.09	1.09	2.55	3.27	4.62	3	2.40	16.14	6.81
DefensiveKV (Feng et al., 2026a)	68.77	32.71	0.81	1.93	2.29	2.18	2.18	3.27	4.74	2	1.20	19.46	6.77
Zip (Shi et al., 2025)	45.28	25.51	0.91	2.51	2.54	0.73	2.91	4.73	1.54	4	3.40	24.15	3.64
CodeComp (Chen et al., 2026)	56.19	31.29	1.09	2.68	3.65	1.09	1.82	4.00	2.67	**6**	3.60	15.73	5.77
SynKV (Ours)	**71.67**	**60.45**	**3.57**	**10.30**	**13.87**	**2.91**	**9.09**	**11.64**	**8.74**	**6**	**22.2**	**65.92**	**48.57**
Δ vs. SOTA	+2.90	+27.74	+2.04	+7.62	+10.22	+0.73	+6.18	+6.91	+4.00	0.00	+18.60	+41.77	+41.14
DeepSeek-Coder-V2-Lite (DS2), 10% Cache Budget (<1% for LB-RP)									512K tokens for LCB and 128K tokens for RB				
Full Cache	76.59	73.22	4.16	6.27	8.13	13.82	17.09	23.27	OOM	OOM	OOM	OOM	OOM
StreamingLLM (Xiao et al., 2024)	34.98	23.42	0.81	0.93	1.12	0.00	0.73	1.45	0.12	0	0.20	5.66	1.34
SnapKV (Li et al., 2024)	36.65	22.51	0.92	0.99	1.08	0.00	0.73	1.09	0.18	1	0.30	7.04	1.27
DuoAttention (Xiao et al., 2025)	39.29	28.19	1.03	1.05	1.33	0.00	1.09	1.82	OOM	OOM	0.20	6.14	3.15
DefensiveKV (Feng et al., 2026a)	45.54	31.60	0.97	0.99	1.05	0.36	1.09	1.45	OOM	OOM	0.20	3.77	1.75
Zip (Shi et al., 2025)	40.16	28.54	0.70	1.03	1.85	0.00	0.73	2.18	OOM	OOM	OOM	OOM	OOM
CodeComp (Chen et al., 2026)	38.25	29.30	1.07	1.15	2.73	0.00	1.45	1.45	OOM	OOM	OOM	OOM	OOM
SynKV (Ours)	**69.59**	**60.07**	**3.29**	**8.39**	**8.39**	**4.00**	**5.82**	**7.27**	**5.56**	**3**	**40.39**	**46.00**	**62.82**
Δ vs. SOTA	+24.05	+28.47	+2.22	+7.24	+5.66	+3.64	+4.37	+5.09	+5.38	+2.00	+40.09	+38.96	+59.67

incurs 1.4–3.6× longer TTFT due to block relevance estimation, it reduces TBT to 0.4× of baseline latency by reusing selected KV blocks across tokens within the same block, making the advantage increasingly significant for long code generation. Zip (Shi et al., 2025) and CodeComp (Chen et al., 2026) introduce considerable prefilling overhead, as PPL-based prompt compression computes perplexity for each chunk, requiring forward computation over nearly the full prompt. Figure 7 further breaks down SynKV's runtime overhead across key components. Critical Block Selection dominates the overhead, requiring 3.0× more time than Syntax-Aligned Block Abstraction, whereas Structural Candidate Retrieval introduces negligible overhead due to efficient hierarchy-based search.

5.4 Ablation Studies

Model Generality. To evaluate SynKV's generality across models, we extend our experiments to additional code LLM backbones, including StarCoder (Li et al., 2023) and CodeLlama (Rozière et al., 2024). In Figure 8, SynKV consistently maintains comparable performance preservation across different model architectures, with the performance degradation relative to full attention varying by less than 2.2% on average. This result demonstrates that SynKV captures general structural attention patterns in long-context code generation rather than relying on architecture-specific behaviors.

Component Ablation. We evaluate the contributions of two key components in SynKV: Structural Candidate Retrieval and Critical Block Selection. First,we replace Structural Candidate Retrieval with a conventional neighboring-block retrieval strategy that selects the three adjacent blocks (*Abl1*). We further ablate Critical Block Selection by replacing adaptive block selection with fixed-size 128-token selection (*Abl2*). Finally, *Abl3* broadens relevance estimation from prefix summaries to full-block scoring, testing whether the prefix alone captures sufficient relevance information. As shown in Figure 9, Abl1 suffers substantial performance degradation on EB and RB, which primarily evaluate code dependency, with an average performance drop of about 45%, while the impact is less pronounced on LB and LCB, which mainly measure surface-level similarity. In contrast, Abl2 causes significant performance degradation across nearly all benchmarks, as fixed token boundaries may disrupt structures for generation. Notably, full-block scoring in Abl3 improves the average score by less than 2%, supporting the representativeness of compact block prefixes.

6 Conclusion

This work addresses efficient long-context code generation by uncovering syntax-aligned block-level attention patterns. Building on this insights, SynKV turns program structure into a practical guide for critical KV retention through syntax-aligned block abstraction. It further streamlines critical KV selection through dependency retrieval and lightweight summarization. Comprehensive evaluations show that SynKV sustains efficient inference while preserving generation quality even under ultra-long contexts and aggressive KV budgets. We hope these findings inspire broader use of semantically organized attention structure in LLM-based generation.

AI USE STATEMENT

In this work, we used generative AI tools to refine the wording of selected sentences. We have not used generative AI tools to generate experimental results, perform data analysis, or make scientific claims, and other disclosure categories are not applicable to this work. The majority of the manuscript was written by the authors. All AI-assisted text was carefully reviewed and revised for accuracy, clarity, originality, and appropriateness. We take full responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

To make SynKV's design and empirical evidence independently inspectable, we provide an anonymized supplementary artifact at `https://anonymous.4open.science/r/SynKV`. Alongside the SynKV implementation, the artifact provides materials for the motivation section: token-indexed attention matrices and figures for examining intra-block coherence, inter-block attention consistency, and block-prefix representativeness. It also includes the evaluation setup and implementations for comparisons with state-of-the-art KV-compression systems, sample benchmark inputs, representative results, and execution instructions. Experiments were conducted on NVIDIA A800 GPUs, using Python 3.10, PyTorch 2.4, and CUDA 12.4.

REFERENCES

Jinwoo Ahn, Ingyu Seong, Akhil Kedia, Junhan Kim, Hyemi Jang, Kangwook Lee, and Yongkweon Jeon. LookaheadKV: Fast and accurate KV cache eviction by glimpsing into the future without generation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=RVLMGPXt2i.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 4895–4901, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.298. URL https://aclanthology.org/2023.emnlp-main.298/.

Ahmed Nusayer Ashik, Shaowei Wang, Tse-Hsun Chen, Muhammad Asaduzzaman, and Yuan Tian. When llms lag behind: Knowledge conflicts from evolving apis in code generation, 2026. URL https://arxiv.org/abs/2604.09515.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench: A bilingual, multitask benchmark for long context understanding. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3119–3137, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.172. URL https://aclanthology.org/2024.acl-long.172/.

Qiujiang Chen, Jing Xiong, Chenyang Zhao, Sidi Yang, and Ngai Wong. Codecomp: Structural kv cache compression for agentic coding, 2026. URL https://arxiv.org/abs/2604.10235.

Xiancai Chen, Zhengwei Tao, Kechi Zhang, Changzhi Zhou, Xinyu Zhang, Wanli Gu, Yuanpeng He, Mengdi Zhang, Xunliang Cai, Haiyan Zhao, and Zhi Jin. Revisit self-debugging with self-generated tests for code generation. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 18003–18023, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.881. URL https://aclanthology.org/2025.acl-long.881/.

Giulio Corallo and Paolo Papotti. FINCH: Prompt-guided key-value cache compression for large language models. *Transactions of the Association for Computational Linguistics*, 12:1517–1532, 2024. doi: 10.1162/tacl_a_00716. URL https://aclanthology.org/2024.tacl-1.83/.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence, 2024. URL https://arxiv.org/abs/2406.11931.

Yuan Feng, Haoyu Guo, Junlin Lv, S Kevin Zhou, and Xike Xie. DefensiveKV: Taming the fragility of KV cache eviction in LLM inference. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=nJgS06sX3O.

Yuan Feng, Junlin Lv, Haoyu Guo, Yukun Cao, S Kevin Zhou, and Xike Xie. CriticalKV: Optimizing KV cache eviction from an output perturbation perspective. In *Forty-third International Conference on Machine Learning*, 2026b. URL https://openreview.net/forum?id=vMwlcLyI4b.

Yu Fu, Zefan Cai, Abedelkadir Asi, Wayne Xiong, Yue Dong, and Wen Xiao. Not all heads matter: A head-level KV cache compression method with integrated retrieval and reasoning. In

The Thirteenth International Conference on Learning Representations, 2025. URL https://openreview.net/forum?id=FJFVmeXusW.

Wenhao Gao, Haoran Cao, Yueyan Li, YongGao Xiao, Caixia Yuan, and Xiaojie Wang. EntroKV: Entropy-guided dynamic budget allocation for KV-cache compression. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=xhAMjsnWUe.

Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. Model tells you what to discard: Adaptive KV cache compression for LLMs. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=uNrFpDPMyo.

Shansan Gong, Ruixiang ZHANG, Huangjie Zheng, Jiatao Gu, Navdeep Jaitly, Lingpeng Kong, and Yizhe Zhang. Diffucoder: Understanding and improving masked diffusion models for code generation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=58NA3unZj5.

Jitai Hao, Yuke Zhu, Tian Wang, Jun Yu, Xin Xin, Bo Zheng, Zhaochun Ren, and Sheng Guo. OmniKV: Dynamic context selection for efficient long-context LLMs. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=ulCAPXYXfa.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024. URL https://arxiv.org/abs/2409.12186.

Huiqiang Jiang, YUCHENG LI, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=fPBACAbqSN.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.*, 35(2), January 2026. ISSN 1049-331X. doi: 10.1145/3747588. URL https://doi.org/10.1145/3747588.

Jang-Hyun Kim, Jinuk Kim, Sangwoo Kwon, Jae W. Lee, Sangdoo Yun, and Hyun Oh Song. KVzip: Query-agnostic KV cache compression with context reconstruction. In *ES-FoMo III: 3rd Workshop on Efficient Systems for Foundation Models*, 2025. URL https://openreview.net/forum?id=gcqzyyF654.

Minsoo Kim, Arnav Kundu, Han-Byul Kim, Richa Dixit, and Minsik Cho. Epicache: Episodic KV cache management for long-term conversation on resource-constrained environments. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=DzvMz3I5HV.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Trans. Softw. Eng. Methodol.*, 34(2), January 2025. ISSN 1049-331X. doi: 10.1145/3690635. URL https://doi.org/10.1145/3690635.

Raymond Li, Loubna Ben allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Ben Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason T Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Urvashi Bhattacharyya, Wenhao Yu, Sasha Luccioni, Paulo Villegas, Fedor Zhdanov, Tony Lee, Nadav Timor, Jennifer Ding, Claire S Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite,

Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro Von Werra, and Harm de Vries. Starcoder: may the source be with you! *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=KoFOg41haE. Reproducibility Certification.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. SnapKV: LLM knows what you are looking for before generation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=poE54GOq2l.

Ren-Biao Liu, Anqi Li, Chaoding Yang, Hui Sun, and Ming Li. Revisiting chain-of-thought in code generation: Do language models need to learn reasoning before coding? In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=wSZeQoJ1Vk.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=pPjZIOuQuF.

Weian Mao, Xi Lin, Wei Huang, Yuxin Xie, Tianfu Fu, Bohan Zhuang, Song Han, and Yukang Chen. Triattention: Efficient long reasoning with trigonometric KV compression. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=0tgzJK50Jz.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. Repograph: Enhancing AI software engineering with repository-level code graph. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=dw9VUsSHGB.

Akshat Ramachandran, Marina Neseem, Charbel Sakr, Rangharajan Venkatesan, Brucek Khailany, and Tushar Krishna. ThinKV: Thought-adaptive KV cache compression for efficient reasoning models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=M3CeHnZKNC.

Stefano Rando, Luca Romani, Alessio Sampieri, Luca Franco, John Yang, Yuta Kyuragi, Fabio Galasso, and Tatsunori Hashimoto. Longcodebench: Evaluating coding LLMs at 1m context windows. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=GFPoM8Ylp8.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

Yuling Shi, Yichun Qian, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. Longcodezip: Compress long context for code language models. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 141–153, 2025. doi: 10.1109/ASE63991.2025.00020.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: query-aware sparsity for efficient long-context llm inference. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

KaiXin Wang, Tianlin Li, Xiaoyu Zhang, Aishan Liu, Xianglong Liu, Ziqi Liu, Zhiqiang Zhang, JUN ZHOU, and Bin Shi. Codechemist: Test-time scaling for low-resource code generation via functional knowledge transfer. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=WVm7jMnA9t.

Zihao Wang, Bin CUI, and Shaoduo Gan. Squeezeattention: 2d management of KV-cache in LLM inference via layer-wise optimal budget. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=9HK2rHNAhd.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 21875–21895, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/5e5fd18f863cbe6d8ae392a93fd271c9-Paper-Conference.pdf.

Guangxuan Xiao, Jiaming Tang, Jingwei Zuo, Junxian Guo, Shang Yang, Haotian Tang, Yao Fu, and Song Han. Duoattention: Efficient long-context llm inference with retrieval and streaming heads. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 37228–37253, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/5c1ddd2e59df46fd2aa85c833b1b36ed-Paper-Conference.pdf.

Jing Xiong, Jianghan Shen, Chuanyang Zheng, Zhongwei Wan, Chenyang Zhao, Chiwun Yang, Fanghua Ye, Hongxia Yang, Lingpeng Kong, and Ngai Wong. Parallelcomp: Parallel long-context compressor for length extrapolation. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=BiM4ewH9zD.

Yuhui Xu, Zhanming Jie, Hanze Dong, Lei Wang, Xudong Lu, Aojun Zhou, Amrita Saha, Caiming Xiong, and Doyen Sahoo. Think: Thinner key cache by query-driven pruning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=n0OtGl6VGb.

Shang Yang, Junxian Guo, Haotian Tang, Qinghao Hu, Guangxuan Xiao, Jiaming Tang, Yujun Lin, Zhijian Liu, Yao Lu, and Song Han. LServe: Efficient long-sequence LLM serving with unified sparse attention. In *Eighth Conference on Machine Learning and Systems*, 2025. URL https://openreview.net/forum?id=KQ4pJFFqT1.

Xiao Yu, Yi Yu, Dongrui Liu, Kejiang Chen, Weiming Zhang, Nenghai Yu, and Jing Shao. EvoBench: Towards real-world LLM-generated text detection benchmarking for evolving large language models. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 14605–14620, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.754. URL https://aclanthology.org/2025.findings-acl.754/.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. Large language models meet NL2Code: A survey. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7443–7464, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.411. URL https://aclanthology.org/2023.acl-long.411/.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Re, Clark Barrett, Zhangyang Wang, and Beidi Chen. H2o: Heavy-hitter oracle for efficient generative inference of large language models. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=RkRrPp7GKO.

A Appendix

A.1 Structural Explanation of Intra-Block Attention Coherence

In code generation, syntax-aligned blocks often form locally coherent program units, where tokens are connected through shared scope, control-flow context, data dependencies, and nearby semantic roles. For example, tokens within the same function body, loop, or conditional branch are typically interpreted under a common program context. Since self-attention learns query-key interactions that reflect such contextual dependencies, tokens inside the same syntax-aligned block tend to assign higher aggregate attention mass to one another than to structurally unrelated tokens outside the block. This provides an attention-level explanation for intra-block coherence.

Specifically, let x_t be a query token and B_i be a syntax-aligned block. We measure the attention relevance between x_t and B_i by aggregating the attention mass from x_t to all key tokens in B_i:

$$M(q_t, B_i) = \sum_{x_j \in B_i} A(q_t, k_j), \quad A(q_t, k_j) = \mathrm{Softmax}\left(\frac{q_t k_j^\top}{\sqrt{d}}\right), \tag{6}$$

where q_t, k_j, and d denote the query vector of x_t, the key vector of x_j, and the attention-head dimension, respectively. When B_i is structurally relevant to x_t, its aggregated attention mass $M(q_t, B_i)$ tends to be larger than that of unrelated blocks:

$$M(q_t, B_i) > M(q_t, B_{\mathrm{unrel}}). \tag{7}$$

Thus, $M(q_t, B_i)$ can be viewed as a learned proxy for the dependency strength between the query token and the syntax-aligned block. This explains why critical query-key interactions frequently concentrate within syntax-aligned blocks and motivates using such blocks as the basic unit for block-level KV retention.

A.2 Additional Overall Performance Analysis

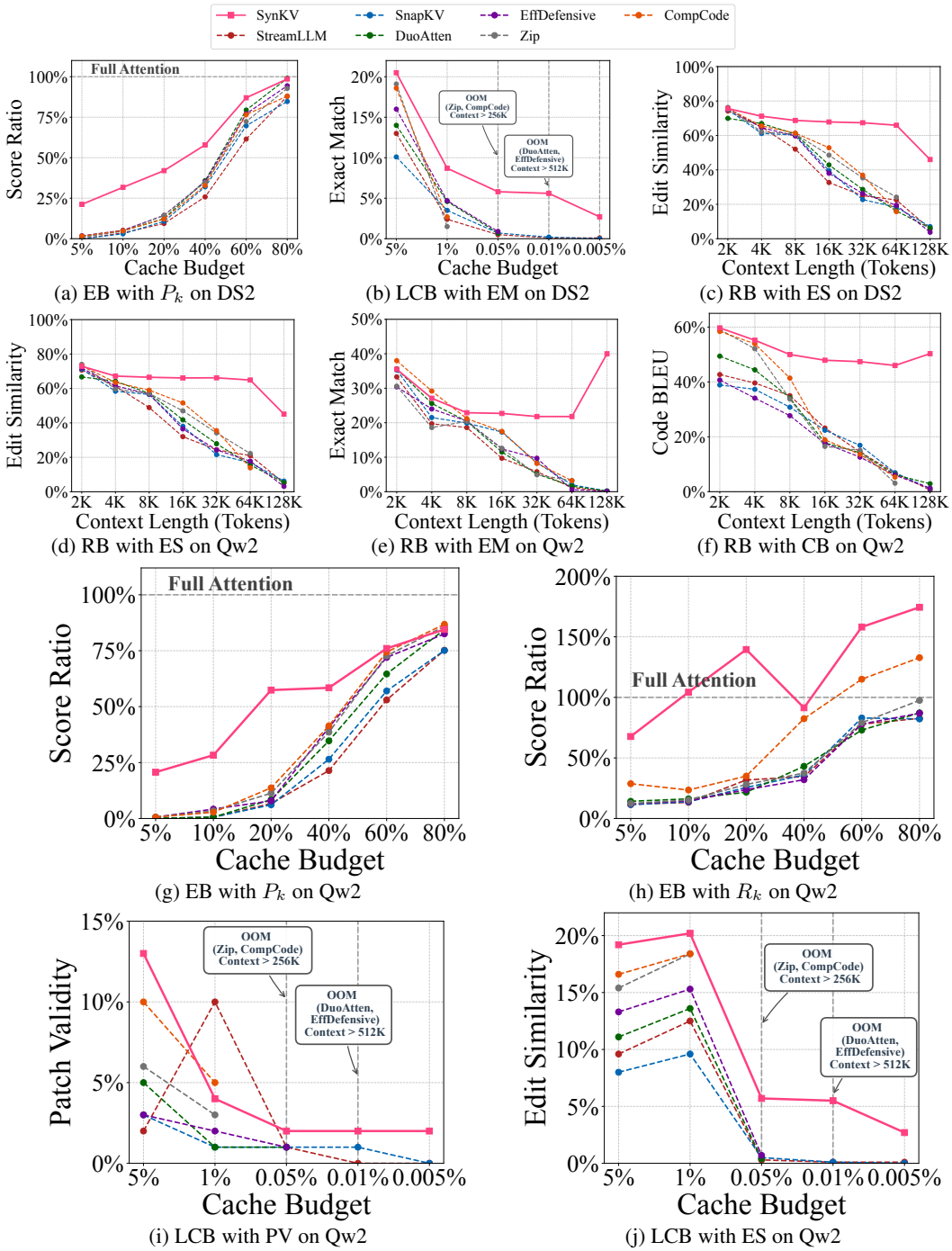
Figure 10 provides a more detailed view of SynKV's overall performance, primarily covering the Qwen2.5-Coder-7B model and additional metrics, including Pass@k and Exact Match, across EB, LCB, and RB. These results complement the main evaluation by showing that SynKV maintains stable generation quality across different models, benchmarks, metrics, and KV-retention ratios, with the advantage becoming particularly evident as context length and memory pressure increase.

An interesting trend appears for CodeBLEU when compared with full attention. At relatively moderate compression ratios, SynKV can outperform full attention by filtering less relevant context and concentrating attention on structurally related code regions, thereby strengthening the information most useful for generation. As compression becomes more aggressive, however, the loss of useful context gradually dominates this benefit, causing SynKV to fall below full attention. This transition highlights the trade-off between structure-aware context concentration and information loss under increasingly constrained KV budgets.

A.3 Additional Performance Analysis across Metrics

Figure 11 presents the corresponding metric-level results on the Qwen2.5-Coder-7B model, complementing the DeepSeek-Coder-V2-Lite results reported in Figure 6. Across the six reported dimensions, SynKV exhibits a performance profile consistent with DeepSeek-Coder-V2-Lite, maintaining clear advantages over existing methods despite differences in model backbone. The largest gains again appear in long-context surface similarity, semantic correctness, dependency awareness, and syntactic validity, where preserving structurally relevant code regions becomes increasingly important. In contrast, the margins remain narrower for typical-context surface similarity and functional correctness, consistent with the DeepSeek-Coder-V2-Lite results: short-range repetitive patterns reduce the sensitivity of surface-level metrics, while functional correctness remains more strongly constrained by the underlying model capability.

Comparing the two models further shows that the relative strengths of individual methods vary across metrics, whereas SynKV preserves a consistently strong profile across both backbones. In particular, its advantage in dependency-aware and semantic-oriented metrics remains pronounced,



(a) EB with P_k on DS2 (b) LCB with EM on DS2 (c) RB with ES on DS2

(d) RB with ES on Qw2 (e) RB with EM on Qw2 (f) RB with CB on Qw2

(g) EB with P_k on Qw2 (h) EB with R_k on Qw2

(i) LCB with PV on Qw2 (j) LCB with ES on Qw2

Figure 10: Detailed overall performance on EB, LCB, and RB under varying KV-retention ratios, including Pass@k and Exact Match on Qw2 and DS2 models.

indicating that syntax-aligned retention generalizes beyond a specific model architecture. Overall, the Qwen2.5-Coder-7B results reinforce the main-body observations on DeepSeek-Coder-V2-Lite and demonstrate that SynKV's gains are sustained across models as well as across complementary dimensions of code-generation quality.

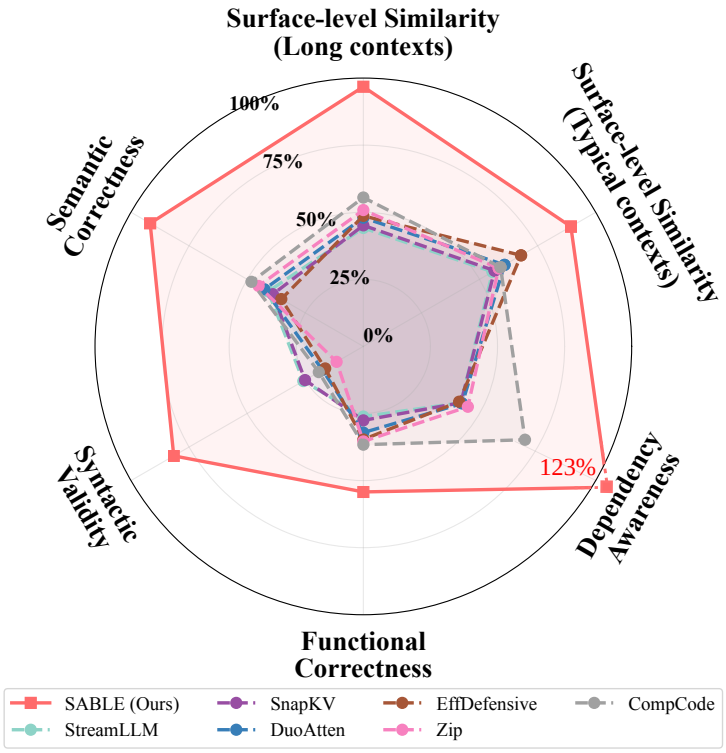


Figure 11: Performance across six metrics on Qw2.

Table 2: Additional performance comparison under different KV retention ratios (%).

Method	LB-LCC ES	LB-RP ES	EVO R_1	EVO R_3	EVO R_5	EVO P_1	EVO P_3	EVO P_5	LCB ES	LCB PV	RB EM	RB ES	RB CB
Qwen2.5-Coder-7B (Qw2), 20% Cache Budget (2% for LB-RP)									128K tokens for LCB and 64K tokens for RB				
Full Cache	73.00	69.69	4.02	6.12	8.01	12.73	14.91	20.36	OOM	OOM	35.07	68.12	54.27
StreamingLLM (Xiao et al., 2024)	58.56	17.73	0.63	0.81	0.88	0	0	0.36	12.54	**10**	1.18	20.98	6.81
SnapKV (Li et al., 2024)	57.61	19.11	0.69	0.84	0.85	0	0	0.36	9.64	1	2.04	16.90	6.87
DuoAttention (Xiao et al., 2025)	62.14	26.48	0.86	0.92	0.97	0	0	0.36	13.57	1	1.62	15.63	6.17
DefensiveKV (Feng et al., 2026a)	66.75	28.08	0.81	0.81	0.82	0	1.09	1.09	15.29	2	0.59	17.76	6.00
Zip (Shi et al., 2025)	42.62	22.23	0.45	0.78	1.73	0	0.73	1.09	18.40	3	3.21	22.16	3.09
CodeComp (Chen et al., 2026)	54.01	26.85	0.96	0.96	2.48	0	0.73	0.73	18.41	5	3.17	13.85	5.35
SynKV (Ours)	**68.67**	**57.31**	**3.14**	**8.10**	**8.20**	**2.55**	**4.36**	**7.27**	**20.25**	4	**21.8**	**64.88**	**45.96**
Δ vs. SOTA	+1.92	+29.23	+2.18	+7.14	+5.72	+2.55	+3.27	+6.18	+1.84	-6.00	+18.59	+42.72	+39.09
Qwen2.5-Coder-7B (Qw2), 10% Cache Budget (<1% for LB-RP)									512K tokens for LCB and 128K tokens for RB				
Full Cache	73.00	69.69	4.02	6.12	8.01	12.73	14.91	20.36	OOM	OOM	OOM	OOM	OOM
StreamingLLM (Xiao et al., 2024)	32.61	19.77	0.57	0.72	0.74	0	0	0	0.11	0	0.10	5.11	0.63
SnapKV (Li et al., 2024)	34.37	17.69	0.60	0.66	0.69	0	0	0	0.14	1	0.20	6.25	1.08
DuoAttention (Xiao et al., 2025)	36.01	24.10	0.72	0.80	0.92	0	0	0	OOM	OOM	0.20	5.34	2.89
DefensiveKV (Feng et al., 2026a)	43.12	28.36	0.57	0.76	0.77	0	0	0.36	OOM	OOM	0.10	3.10	1.32
Zip (Shi et al., 2025)	37.05	24.86	0.50	0.83	0.90	0	0	0.36	OOM	OOM	OOM	OOM	OOM
CodeComp (Chen et al., 2026)	35.38	24.83	1.51	1.60	1.80	0	0	0.36	OOM	OOM	OOM	OOM	OOM
SynKV (Ours)	**66.54**	**55.27**	**2.33**	**4.74**	**5.42**	0	**4.73**	**6.18**	**5.53**	**2**	**40.04**	**45.08**	**50.35**
Δ vs. SOTA	+23.42	+26.91	+0.82	+3.14	+3.62	0.00	+4.73	+5.82	+5.39	+1.00	+39.84	+38.83	+47.46

A.4 Extended Performance Analysis

Table 2 extends the detailed comparison in Table 1 from DeepSeek-Coder-V2-Lite to Qwen2.5-Coder-7B under the same KV-retention settings and long-context configurations. Across both 20% and 10% cache budgets, SynKV preserves the performance advantage observed on DeepSeek-Coder-V2-Lite, indicating that its benefit is not tied to a particular code-LLM backbone. At 20% retention, SynKV reaches 68.67% and 57.31% ES on LB-LCC and LB-RP, respectively, while maintaining substantially stronger performance on dependency-sensitive benchmarks such as RB. Under the more aggressive 10% budget, this gap becomes more pronounced: SynKV retains 66.54% ES on LB-LCC and achieves 40.04% EM, 45.08% ES, and 50.35% CodeBLEU on RB, whereas most baselines collapse to near-zero or single-digit performance.

The Qwen2.5-Coder-7B results also reveal a similar long-context robustness trend to DeepSeek-Coder-V2-Lite. At 512K tokens on LCB, SynKV remains executable and achieves 5.53% ES and 2% patch validity, while several stronger baselines encounter OOM and the remaining methods degrade sharply. A comparable pattern appears on RB at 128K tokens, where SynKV retains meaningful generation quality despite the highly constrained cache budget. Although the absolute scores vary across models—for example, Qwen2.5-Coder-7B shows stronger LCB performance at the 20% budget, while DeepSeek-Coder-V2-Lite generally performs better on several LB and RB metrics—the relative advantage of SynKV remains consistent.

Overall, the additional Qwen2.5-Coder-7B results reinforce two observations from the main table: SynKV degrades more gracefully as the KV budget shrinks, and its advantage becomes increasingly evident as context length grows. This consistency across model backbones suggests that syntax-aligned retention captures structural regularities that generalize beyond model-specific attention behavior.

A.5 Detailed Inference Latency

Table 3: Inference latency comparison across different context lengths.

Method	4K	8K	16K	32K	64K	128K	256K
Total Latency (s) ↓							
StreamingLLM (Xiao et al., 2024)	48.18	48.08	49.26	51.31	56.83	63.68	104.65
SnapKV (Li et al., 2024)	49.81	48.08	48.36	50.86	57.20	64.33	106.72
DuoAttention (Xiao et al., 2025)	48.57	48.16	47.48	58.87	60.36	64.60	107.81
DefensiveKV (Feng et al., 2026a)	50.99	50.66	49.67	59.66	62.01	66.03	106.83
Zip (Shi et al., 2025)	54.18	86.12	173.62	276.95	355.65	563.77	OOM
CodeComp (Chen et al., 2026)	49.61	53.33	62.51	81.13	103.28	134.53	OOM
SynKV (Ours)	**23.02**	**30.34**	**30.62**	**37.90**	**49.27**	**57.23**	**99.40**
Δ vs. SOTA	-25.16	-17.74	-16.86	-12.96	-7.56	-6.45	-5.25
Time to First Token (TTFT, s) ↓							
StreamingLLM (Xiao et al., 2024)	0.452	0.578	1.055	2.482	**5.020**	**11.151**	**23.227**
SnapKV (Li et al., 2024)	0.476	0.581	1.052	2.483	5.070	13.459	24.729
DuoAttention (Xiao et al., 2025)	0.464	**0.577**	**1.039**	3.237	6.828	13.086	24.772
DefensiveKV (Feng et al., 2026a)	0.523	0.634	1.087	**2.479**	6.612	14.947	26.893
Zip (Shi et al., 2025)	**0.311**	0.599	1.559	3.436	18.861	64.081	OOM
CodeComp (Chen et al., 2026)	0.883	0.972	1.782	3.233	15.459	52.613	OOM
SynKV (Ours)	1.282	2.310	4.145	9.196	17.511	25.139	63.039
Δ vs. SOTA	+0.971	+1.733	+3.106	+6.717	+12.491	+13.988	+39.812
Time Between Tokens (TBT, s) ↓							
StreamingLLM (Xiao et al., 2024)	0.093	0.093	0.094	0.098	0.093	0.099	0.097
SnapKV (Li et al., 2024)	0.096	0.093	0.092	0.097	0.095	0.107	0.102
DuoAttention (Xiao et al., 2025)	0.094	0.093	0.094	0.096	0.104	0.104	0.184
DefensiveKV (Feng et al., 2026a)	0.099	0.098	0.095	0.098	0.114	0.123	0.198
Zip (Shi et al., 2025)	0.054	**0.053**	**0.055**	0.134	0.189	0.314	OOM
CodeComp (Chen et al., 2026)	0.074	0.079	0.086	0.087	0.123	0.159	OOM
SynKV (Ours)	**0.045**	0.060	0.059	**0.062**	**0.060**	**0.062**	**0.062**
Δ vs. SOTA	-0.009	+0.007	+0.004	-0.025	-0.033	-0.037	-0.035

Table 3 provides the detailed latency breakdown across context lengths from 4K to 256K tokens, complementing the aggregate results in Figure 5. SynKV achieves the lowest total latency across all evaluated context lengths. At 4K tokens, it requires only 23.02 s, reducing latency by 52.2% over the fastest baseline (48.18 s). The reduction remains 36.9%, 35.5%, and 25.5% at 8K, 16K, and 32K tokens, respectively. At longer contexts, SynKV still reduces total latency by 13.3% at 64K and 10.1% at 128K. Even at 256K tokens, where relevance-estimation overhead becomes increasingly pronounced, SynKV completes generation in 99.40 s compared with 104.65 s for the fastest baseline.

The latency decomposition further reveals a clear trade-off between prefilling and decoding. SynKV incurs additional TTFT for syntax-aware relevance estimation, increasing from 1.282 s at 4K to 63.039 s at 256K. In contrast, its TBT remains remarkably stable, ranging only from 0.045 s to

0.062 s over the entire 4K–256K range. Compared with the fastest general KV-compression baseline at each context length, SynKV reduces TBT by approximately 35–52%. At 128K, for example, SynKV achieves a TBT of 0.062 s versus 0.099 s for StreamingLLM, while at 256K the corresponding values are 0.062 s and 0.097 s. This behavior confirms that reusing selected KV blocks within the same syntax block effectively amortizes selection decisions across generated tokens, preventing decoding cost from growing substantially with context length.

Prompt-compression baselines exhibit a different scaling behavior. At 128K tokens, Zip and CodeComp require 563.77 s and 134.53 s in total, respectively, compared with 57.23 s for SynKV, corresponding to approximately $9.9\times$ and $2.4\times$ higher latency. Their prefilling overhead also grows rapidly with context length: Zip reaches 64.08 s TTFT and CodeComp 52.61 s at 128K, while both fail with OOM at 256K. These results reinforce that query-conditioned PPL evaluation becomes increasingly costly for long prompts, whereas SynKV trades moderate relevance-estimation overhead for substantially lower and more stable decoding latency.