CODECOMP: STRUCTURAL KV CACHE COMPRESSION FOR AGENTIC CODING

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic coding systems repeatedly read source files, execute tools, and contain long interaction histories, making KV cache a major bottleneck for repository-level inference. Existing KV cache compression methods largely rely on attention signals, which can miss code tokens that are structurally important for program understanding. We present **CodeComp**, a training-free KV cache compression method for agentic coding that incorporates static program structure into cache retention. CodeComp uses Joern-derived structural annotations to guide compression and preferential retention within historical READ results. On a subset of SWE-bench Verified, CodeComp achieves resolve rates close to those of the native agent on both Qwen3.6-27B and Devstral-Small-2-24B, while reducing peak KV cache memory during inference by 22.1% and 10.6%, respectively. These findings support program structure as a complementary signal for KV cache compression. Code: `https://anonymous.4open.science/r/CodeComp/`.

1 INTRODUCTION

Repository-level coding agents interact with a repository over multiple turns to inspect code, run tests, and refine patches (Jimenez et al., 2024; Xia et al., 2024; Yao et al., 2023; Yang et al., 2024). Each interaction extends the history included in subsequent model calls, increasing the context that the model must process. During inference, the KV cache stores key and value states for this accumulated context. Its memory footprint grows as the trajectory lengthens and can become a major inference-memory bottleneck (Liu et al., 2024; Hooper et al., 2024).

KV cache compression commonly uses quantization or selective retention. Quantization methods such as KIVI (Liu et al., 2024) and KVQuant (Hooper et al., 2024) reduce cache precision while retaining all tokens. CodeComp follows selective retention, whose methods differ in their relevance signals and selection granularity. H2O (Zhang et al., 2023) accumulates attention to retain heavy-hitter tokens, SnapKV (Li et al., 2024) estimates importance from an observation window, Quest (Tang et al., 2024b) selects relevant KV cache pages using the current query, and ChunkKV (Liu et al., 2025) retains attention-ranked token chunks to preserve local context.

These methods provide effective mechanisms for identifying relevant tokens or locally coherent regions. In coding-agent trajectories, however, selection operates over a growing history that combines task instructions, recent interactions, and previously retrieved source code. This setting raises two additional questions: which parts of the trajectory should be compressed, and how should retention account for the program structure within retrieved code?

The first challenge is the scope of compression. The accumulated context contains the task instructions, the agent's recent actions, and results from earlier tool calls; removing content from these parts of the history has different consequences. In particular, code returned by READ can persist across many later model calls, even after the agent has moved on to other files or started testing a patch. In our analysis, READ results account for 41.65% of cumulative prompt-token occurrences, the largest share of any content category (§2.1). This concentration in historical code suggests a focused compression target while allowing task instructions and interactions to remain intact.

The second challenge is determining which tokens within retrieved code can be evicted. Attention provides a query-conditioned estimate of relevance, but token-level scores do not explicitly represent the syntactic and dependency relationships among code elements. A call, branch, or assignment

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.

spans multiple tokens whose roles depend on the surrounding construct. Ranking these tokens independently can retain highly attended fragments while removing other tokens needed to represent the same construct. This creates an attention–structure mismatch: query relevance is not equivalent to structural importance within the program. Chunk-based retention preserves local continuity but does not explicitly model these program relationships. Our aggregate analysis and token-level case study in §2 show this mismatch and motivate program structure as a complementary signal for KV cache compression.

To address these challenges, we introduce CodeComp, a training-free KV cache compression method for coding agents. CodeComp focuses compression on historical code returned by READ, while preserving task instructions and recent interactions. It derives program structure from a Code Property Graph (Yamaguchi et al., 2014) constructed by Joern (Joern Developers, 2026) and uses this information in two complementary ways. Selected code regions receive priority within the retention budget, while a soft structural prior is combined with attention to rank the remaining tokens. The retained KV cache therefore reflects both relevance to the current agent step and structural importance within the source code. Implemented on SGLang (Zheng et al., 2024), CodeComp compresses the KV cache after full-prompt prefill without additional model training or changes to the stored messages.

Our contributions are as follows:

- We characterize the contribution of retrieved code to cumulative context usage and examine the limitations of attention-based token selection in preserving code structure (§2).
- We introduce CodeComp, which combines hard protection and a soft structural prior with attention to compress historical READ KV cache under a fixed retention budget (§3).
- We evaluate CodeComp on a subset of SWE-bench Verified tasks (SWE-bench Team, 2024) using Qwen3.6-27B and Devstral-Small-2-24B. Resolve rates remain close to those of the native agent, while peak KV cache memory decreases by 22.1% and 10.6%. Analyses of structural retention and sensitivity to compression parameters are presented in §4.

2 MOTIVATION

As a coding agent explores a repository, tool calls and their results accumulate in the interaction history and remain available to subsequent model calls. Compressing the resulting KV cache raises two related questions. First, which parts of the growing agent trajectory constitute a useful compression target? Second, once such content is identified, what information should guide retention? We first examine where KV cache usage arises in code agent trajectories (§2.1), and then study the mismatch between query-conditioned attention and program structure in retrieved code (§2.2).

2.1 AGENT CONTEXT

A coding task typically involves multiple rounds of interaction with a repository. The context at each round contains the task instructions and the preceding interaction history, including tool calls and their results. As the history grows, content from earlier rounds remains part of subsequent prompts and contributes to their KV cache usage. Identifying a compression target therefore requires examining the composition of this context across turns.

We analyze the prompts from 50 sampled SWE-bench Verified trajectories, aggregating token counts by content category across all model calls. Tokens that recur in subsequent prompts are counted at each occurrence. The average cumulative count is 550,036 tokens per task. As shown in Figure 1, READ results constitute the largest category, averaging 229,090 tokens per

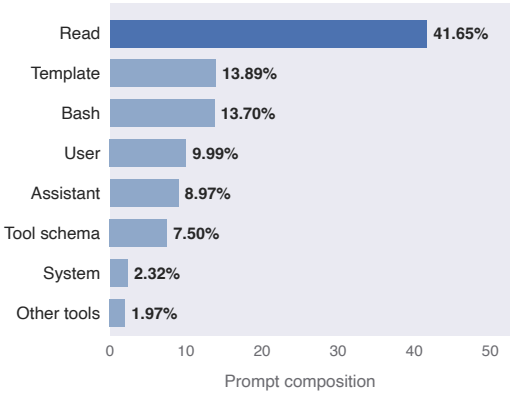
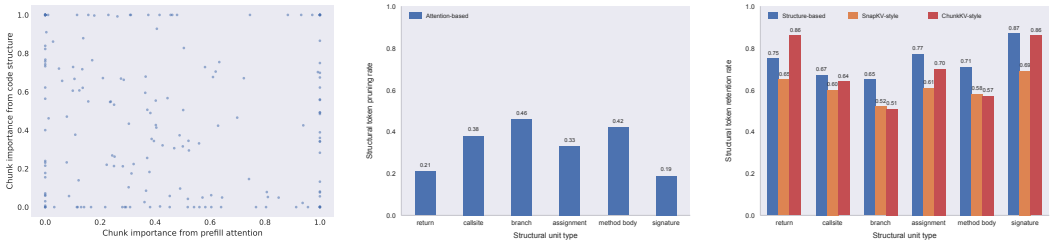


Figure 1: Cumulative prompt composition across agent trajectories.

task and accounting for 41.65% of cumulative prompt tokens. Template text and BASH results are the next largest categories at 13.89% and 13.70%, respectively. User messages, assistant text, tool schemas, system messages, and other tool results each account for less than 10%.

These findings motivate historical READ results as a primary target for KV cache compression. The following subsection examines the limitations of attention-based compression when applied to this code.

2.2 ATTENTION–STRUCTURE MISMATCH



(a) Attention vs. structure (b) Structural mis-pruning (c) Structure-aware retention

Figure 2: Structural analysis on 50 sampled SWE-bench Verified issues. (a) Prefill attention and Joern-derived structural importance assign different importance scores to the same retrieved code regions. (b) Attention-based compression removes structurally important tokens at different rates across program constructs. (c) Under the same KV cache budget, structure-aware selection retains more structurally important tokens in most categories.

Attention-based KV cache methods estimate the relevance of cached tokens using attention from the current query or a recent observation window (Li et al., 2024; Tang et al., 2024b). For retrieved source code, however, attention-based relevance captures only one dimension of importance and may not reflect structural or semantic dependencies among code elements. Calls, branches, and assignments span multiple tokens connected by syntactic, control, and data dependencies. Query-conditioned attention can assign different scores to tokens within the same construct because it ranks tokens by relevance rather than preserving structural boundaries.

This distinction creates an attention–structure mismatch. A code region that contains low-attention tokens may still form part of a call, condition, or assignment whose other tokens receive higher scores. Low attention to an individual position therefore does not establish that it can be removed without fragmenting the surrounding construct.

We examine this mismatch in historical READ results from 50 sampled SWE-bench Verified trajectories using Qwen3.6-27B. Figure 2a shows that high structural importance does not consistently correspond to high prefill attention; some code chunks receive high structural scores despite low attention. Attention alone therefore does not identify all code regions that are important under the structural analysis.

The mismatch also leads to non-uniform pruning across program constructs. As shown in Figure 2b, attention-based compression removes 46% of structurally important branch tokens and 42% of method-body tokens, compared with 21% of return tokens and 19% of signature tokens. Branches and method bodies are therefore pruned most frequently among the examined categories. Figure 2c previews how structural guidance changes retention under the same KV cache budget; we defer the quantitative comparison with SnapKV-style and ChunkKV-style selection to §4.3.

2.3 CASE STUDY

We examine the SWE-bench Verified instance `astropy__astropy-12907` using `Qwen/Qwen3.6-27B`. Figure 3 shows token retention for lines 621–680 of `astropy/modeling/core.py`, retrieved by the agent while solving the task. Under the same 60% retention ratio, Attention Top-K leaves gaps within calls, branches, and assignments, whereas CodeComp retains more coherent portions of these

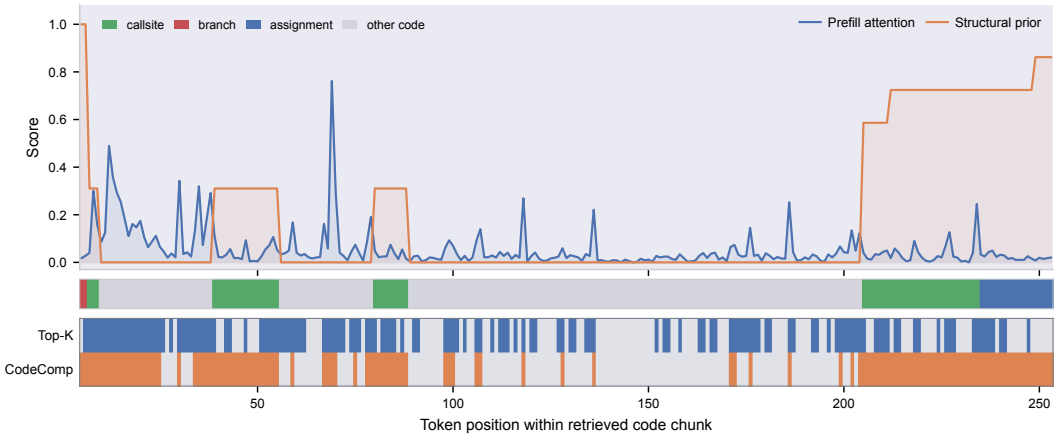


Figure 3: Token-level retention on a retrieved code chunk from the SWE-bench Verified instance `astropy__astropy-12907`. Under the same 60% KV cache retention ratio, Attention Top-K selects isolated high-attention tokens, whereas CodeComp uses structural guidance to preserve more coherent regions associated with call sites, branches, and assignments.

constructs. This example shows the token-level consequence of the aggregate mismatch in Figure 2: locally salient tokens do not necessarily preserve the surrounding program structure.

Together, the context analysis and structural analysis identify historical READ results as a substantial source of KV cache usage and show that query-conditioned attention alone does not reliably preserve the structure of retrieved code. These observations motivate structure-aware KV cache compression for historical code in coding-agent trajectories.

3 STRUCTURE-BASED KV CACHE COMPRESSION

CODECOMP is a training-free KV cache compression method for coding agents. It uses program structure to guide which tokens to retain from the agent's historical trajectory. Figure 4 outlines the method. We first define the compression candidates and the active query window within the trajectory (§3.1). We then derive a protected token set and a soft importance prior from code structure (§3.2). Finally, we combine this guidance with query-conditioned attention to select KV cache under a fixed budget (§3.3).

3.1 AGENT TRAJECTORY

As the agent works on a task, its messages accumulate in the trajectory. Let S be the fixed system prompt and H_{t-1} the trajectory before turn t. If the model produces a tool call in assistant message A_t and receives the corresponding tool result T_t, the trajectory is updated as

$$H_t = H_{t-1} \parallel (A_t, T_t). \tag{1}$$

Each interaction appends new messages without removing earlier ones, so the trajectory grows over successive turns. At each model invocation, CodeComp partitions the serialized prompt into three disjoint token sets: query, compression, and retention. This partition is recomputed at every invocation and does not modify the trajectory itself.

Query. Let $\mathcal{Q}_t$ denote the token occupied by the final q trajectory entries. These tokens are retained in full and supply the query vectors used to score historical candidate KV cache. In the current turn, this window typically contains the latest assistant tool call and its result.

Compression. Let $\mathcal{C}_t$ denote the tokens in historical READ results that fall outside the recent window of size k and can be aligned with the corresponding source code. CodeComp applies KV cache compression only to these tokens.

Retention. Let $\mathcal{K}_t$ denote all non-query tokens outside $\mathcal{C}_t$. These tokens are retained without KV cache compression.

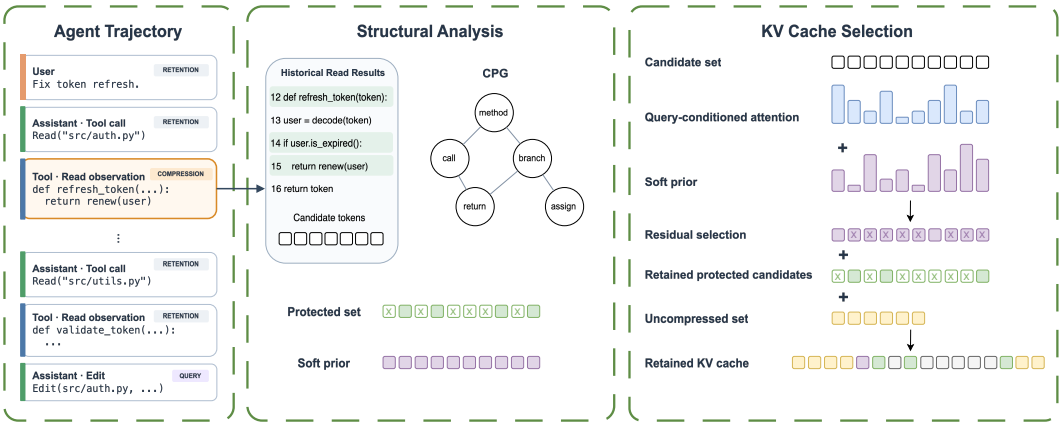


Figure 4: Overview of CODECOMP. CodeComp compresses eligible historical READ results while retaining the remaining context in full. Structural analysis produces a protected set and a soft prior, which is combined with query-conditioned attention to rank candidates. The retention budget is allocated to protected candidates first, then filled from the remaining candidates. The selected KV cache is combined with the uncompressed KV cache for decoding.

3.2 STRUCTURAL ANALYSIS

CodeComp converts source structure into two token-level guidance signals. The protected set $\mathcal{P}_t$ gives selected candidate tokens priority under a fixed retention budget, while the soft prior $\bar{w}_p$ augments the attention-based score used to rank candidates. Both signals are derived from source code snippets and mapped to their corresponding positions in the serialized prompt.

Structural analysis. For each READ result containing source code, CodeComp uses Joern (Joern Developers, 2026) to construct a CPG (Yamaguchi et al., 2014) for the returned source window. It extracts method and type declarations, call sites, branches, loops, returns, and assignments, together with control-flow, control-dependence, and def-use edges. Each extracted program construct is represented as a typed source span $z = (\ell_s, \ell_e, \tau)$, where ℓ_s and ℓ_e denote its first and last source lines and τ denotes its construct type. We assign z the structural score

$$g(z) = g_{\text{type}}(z) + g_{\text{dep}}(z) + g_{\text{task}}(z), \tag{2}$$

where g_{type} scores the syntactic category of z, g_{dep} measures its involvement in CPG dependencies, and g_{task} measures the lexical overlap between its identifiers and the task description. The task term reweights the constructs extracted from the CPG without introducing additional candidates.

Hard and soft structural guidance. CodeComp selects a subset of spans $\mathcal{Z}_t^{\text{prot}}$ for hard protection. For each READ result, hard protection covers at most L_{prot} distinct source lines. CodeComp first selects at most one task-aligned method body whose length does not exceed L_{body}. It then fills the remaining line budget with high-scoring calls, branches, loops, returns, and assignments, followed by method and class signatures not already covered. All extracted spans contribute their scores to the soft prior.

CodeComp then maps each span z to its candidate-token positions $\phi_t(z) \subseteq \mathcal{C}_t$ in the serialized prompt. It locates the corresponding READ result after chat templating and aligns the numbered source lines with token intervals. The protected-token set is

$$\mathcal{P}_t = \bigcup_{z \in \mathcal{Z}_t^{\text{prot}}} \phi_t(z) \subseteq \mathcal{C}_t. \tag{3}$$

Each candidate token $p \in \mathcal{C}_t$ also receives a nonnegative soft structural weight

$$w_p = \max_{z:\, p \in \phi_t(z)} g(z), \tag{4}$$

with $w_p = 0$ when no annotated span covers p. We normalize the prior within the current candidate pool:

$$\bar{w}_p = \frac{w_p}{\max_{p' \in \mathcal{C}_t} w_{p'}}, \tag{5}$$

If a READ result cannot be aligned with the prompt tokens, CodeComp retains it in full. If alignment fails for an individual span, CodeComp discards only the corresponding structural annotation.

3.3 KV CACHE SELECTION

Given the candidate set $\mathcal{C}_t$, protected set $\mathcal{P}_t$, and soft prior $\bar{w}_p$, CodeComp selects which candidate tokens to retain in each attention layer's KV cache.

Token scoring. For each layer ℓ, CodeComp uses attention from $\mathcal{Q}_t$ to assign each candidate $p \in \mathcal{C}_t$ a query-conditioned relevance score $a_\ell(p)$.

CodeComp combines this dynamic relevance signal with the normalized soft structural prior:

$$u_\ell(p) = a_\ell(p) + \alpha \bar{w}_p, \tag{6}$$

where $\alpha \geq 0$ controls the contribution of the soft prior. This score is used to rank tokens within both the protected and non-protected pools.

Budget allocation. Let $N_t = |\mathcal{C}_t|$ be the number of candidate tokens and let $r \in (0, 1]$ be their target retention ratio. CodeComp retains a fixed budget of $B_t = \lceil rN_t \rceil$ candidate tokens.

As illustrated in Figure 4, CodeComp allocates the budget to protected candidates first:

$$B_t^{\text{prot}} = \min(B_t, |\mathcal{P}_t|), \tag{7}$$

$$B_t^{\text{res}} = B_t - B_t^{\text{prot}}. \tag{8}$$

If the protected set fits within the budget, all protected candidates are retained. Otherwise, the entire budget is assigned to the highest-scoring protected candidates. Any remaining slots are filled from non-protected candidates.

KV cache selection. Let $\text{TopK}_k(\mathcal{A}; s)$ return the k elements in set $\mathcal{A}$ with the highest scores under s. The retained protected candidates are

$$\widehat{\mathcal{P}}_{t,\ell} = \text{TopK}_{B_t^{\text{prot}}}(\mathcal{P}_t; u_\ell), \tag{9}$$

and the residual selection is

$$\mathcal{S}_{t,\ell} = \text{TopK}_{B_t^{\text{res}}}(\mathcal{C}_t \setminus \mathcal{P}_t; u_\ell). \tag{10}$$

Let $\mathcal{F}_t = \mathcal{K}_t \cup \mathcal{Q}_t$ denote the tokens retained without compression. The final retained token set for layer ℓ is

$$\mathcal{R}_{t,\ell} = \mathcal{F}_t \cup \widehat{\mathcal{P}}_{t,\ell} \cup \mathcal{S}_{t,\ell}. \tag{11}$$

Selected tokens may differ across layers because their attention patterns differ, but every layer retains exactly $|\mathcal{F}_t| + B_t$ tokens and therefore has the same compacted KV cache length.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

We evaluate CodeComp end to end on 100 SWE-bench Verified tasks (SWE-bench Team, 2024) using Qwen3.6-27B (Qwen Team, 2026) and Devstral-Small-2-24B (Mistral AI, 2025). All methods use the same tasks, agent setup, and inference settings. We implement the agent loop using Nano Claude Code, an earlier version of the CheetahClaws agent harness (CheetahClaws Team, 2026), and use Harbor to manage containerized task environments, agent execution, and evaluation (Harbor Framework Team, 2026). CodeComp is implemented on SGLang v0.5.11 (Zheng et al., 2024), passes structural annotations as request metadata without adding them to the model-visible prompt, and caches CPG analyses by file identity, language, and content. By default, we use a retention ratio of $r = 0.60$ and a soft structural-prior weight of $\alpha = 0.25$. We set the hard-protection line budget to $L_{\text{prot}} = 120$ and the maximum protected method-body length to $L_{\text{body}} = 80$ lines. The compression scope follows §3.1. Ablations and structural retention analysis each use 50 tasks with Qwen3.6-27B.

Because compression can alter subsequent tool calls, each method may produce a different trajectory; we therefore treat prompt length and KV cache use as end-to-end measurements. We report resolved tasks, prompt-token occurrences per task aggregated over all turns, compression of both eligible historical READ KV and the full prompt KV, peak KV cache memory, and end-to-end latency.

Table 1: End-to-end task outcomes and KV cache measurements on 100 SWE-bench Verified tasks per model at $r = 0.60$. Eligible KV cache refers to the historical READ tokens assigned to the compression set $\mathcal{C}_t$; overall KV cache includes the full prompt. Bold indicates the highest resolved count and the lowest peak KV cache memory and end-to-end latency within each model.

Method	Task Outcome		Trajectory	Profiling			Agent E2E (s)
	Resolved (/100)	Empty Outputs	Prompt Tokens / Task	Eligible KV Cache Comp. (%)	Overall KV Cache Comp. (%)	Peak KV Cache Memory (GiB)	
Qwen/Qwen3.6-27B							
Nano Native	55	8	235,946	0%	0%	6.46	391.51
Nano Summary	**58**	5	262,426	0%	0%	6.48	378.19
Random	47	15	172,393	39.99%	14.51%	6.18	311.71
SnapKV	50	10	190,457	39.99%	13.66%	5.63	348.13
ChunkKV	53	15	169,875	39.99%	13.24%	5.26	317.93
CodeComp	54	11	163,122	39.99%	13.04%	**5.03**	**282.83**
CodeComp + Summary	55	8	178,414	39.99%	11.64%	5.05	373.11
mistralai/Devstral-Small-2-24B-Instruct-2512							
Nano Native	30	24	440,001	0%	0%	4.35	238.02
Nano Summary	27	18	660,952	0%	0%	3.75	419.28
Random	26	20	426,440	39.99%	15.69%	3.97	**224.66**
SnapKV	26	22	477,431	39.99%	17.97%	3.98	237.02
ChunkKV	29	22	517,693	39.99%	16.57%	3.95	258.85
CodeComp	**32**	26	422,542	39.99%	15.55%	3.89	247.39
CodeComp + Summary	30	25	622,992	39.99%	11.55%	**3.56**	374.96

Table 2: Ablation of Joern structural guidance in CodeComp at a configured $r = 0.60$. All variants use attention for the remaining KV cache selection; Joern-enabled variants additionally gate candidate eligibility using valid structural annotations. Realized agent trajectories and retained KV amounts can differ across variants.

Variant	Joern-gated eligibility	Hard protection	Soft prior	Resolved (/50)	Empty Outputs	Prompt Tokens / Task	Eligible Read KV Cache Compression (%)	Overall KV Cache Compression (%)	Peak KV Cache Memory (GiB)	Agent E2E (s)
No structural guidance	No	No	No	24	6	180,131	39.99%	15.69%	5.89	283.99
Gated attention-only	Yes	No	No	25	4	161,663	39.99%	15.33%	5.98	258.45
Hard-only ($\alpha = 0$)	Yes	Yes	No	20	6	144,672	39.99%	13.52%	5.40	227.76
Soft-only ($\alpha = 0.25$)	Yes	No	Yes	25	3	156,955	39.99%	14.41%	5.12	252.81
Full CodeComp ($\alpha = 0.25$)	Yes	Yes	Yes	28	3	184,874	39.99%	13.18%	5.34	296.46

4.2 MAIN RESULTS

Table 1 compares CodeComp with the uncompressed agent, three structure-agnostic KV compression methods, and conversation summarization. Random selects tokens without an importance signal; SnapKV selects individual tokens using attention from an observation window (Li et al., 2024), while ChunkKV retains attention-ranked token chunks to preserve local context (Liu et al., 2025). We also evaluate CodeComp combined with Nano Claude Code's built-in conversation summarization.

Native agent. CodeComp reduces peak KV cache memory while resolving a similar number of tasks to the native agent. Peak memory decreases from 6.46 to 5.03 GiB on Qwen3.6-27B and from 4.35 to 3.89 GiB on Devstral-Small-2-24B. The corresponding resolved counts are 54 versus 55 and 32 versus 30: compression slightly reduces task resolution on Qwen3.6-27B, while the higher count on Devstral-Small-2-24B indicates that it may also improve resolution. The latency benefit is model-dependent. Qwen3.6-27B's end-to-end latency decreases from 391.51 to 282.83 s, whereas Devstral-Small-2-24B's increases from 238.02 to 247.39 s.

KV compression methods. Among the KV compression methods, CodeComp resolves the most tasks and uses the least peak KV cache memory on both models. All methods compress 39.99% of eligible KV cache, but their task resolution rates differ. Random and SnapKV resolve 47 and 50 tasks on Qwen3.6-27B and 26 tasks each on Devstral-Small-2-24B. ChunkKV provides the strongest baseline, resolving 53 and 29 tasks, compared with CodeComp's 54 and 32. CodeComp also reduces peak memory from ChunkKV's 5.26 to 5.03 GiB on Qwen3.6-27B and from 3.95 to 3.89 GiB on Devstral-Small-2-24B. Its advantage over ChunkKV is therefore modest, while the gap in resolved tasks is larger against Random and SnapKV.

Context summarization. Summarization and CodeComp exhibit different trade-offs across the two models. On Qwen3.6-27B, summarization resolves 58 tasks compared with CodeComp's 54, but uses 6.48 GiB of peak KV cache memory compared with 5.03 GiB. On Devstral-Small-2-24B, it uses slightly less memory at 3.75 versus 3.89 GiB, but resolves 27 tasks compared with CodeComp's 32. Combining summarization with CodeComp does not consistently improve either metric: it resolves 55 tasks with 5.05 GiB on Qwen3.6-27B and 30 tasks with 3.56 GiB on Devstral-Small-2-24B, while increasing latency over CodeComp alone on both models. The combination offers an additional memory reduction on Devstral, but this accompanies a lower resolved count.

Structural annotation overhead. The reported end-to-end latency includes Joern annotation, which takes 7.82 s per uncached source result on average and 0.06 s when cached.

4.3 Structural Retention Analysis

We return to the structural analysis introduced in §2.2 and quantify how structural guidance changes the code retained under the same target KV cache size. The comparison covers returns, call sites, branches, assignments, method bodies, and signatures across 50 sampled SWE-bench Verified issues using Qwen3.6-27B.

At the same target KV cache size, structure-aware selection retains a larger fraction of structurally important tokens than the SnapKV-style and ChunkKV-style baselines in five of the six categories in Figure 2c. Branch retention rises to 65%, compared with 52% and 51% for the two baselines; method-body retention reaches 71%, compared with 58% and 57%. These improvements occur in the two categories with the highest removal fractions under attention-based compression. Returns are an exception: ChunkKV-style selection retains 86%, compared with 75% for structure-aware selection. Structural guidance thus improves coverage across most of the examined code categories, although the benefit is not uniform across categories.

4.4 Ablation Studies

We evaluate the retention ratio, soft structural weight, and structural components on the same 50-task subset using Qwen3.6-27B. Complete measurements are reported in Appendix A.

Retention ratio. Lower retention ratios increase KV cache compression, but retaining more tokens does not consistently improve task resolution. Figure 5(a) varies r from 0.20 to 1.00 with $\alpha = 0.25$. At $r = 0.20$, CodeComp compresses 26.99% of the overall KV cache and resolves 25 tasks. Increasing r to 0.60 reduces compression to 14.18% and raises the resolved count to 28, the highest in this comparison. Further increasing r to 0.80 or 1.00 resolves only 24 or 25 tasks. Thus, more aggressive compression reduces the resolved count relative to $r = 0.60$, but the full range does not exhibit a monotonic relationship between retention and task resolution.

Soft structural weight. The number of resolved tasks is sensitive to the soft structural weight. Figure 5(b) varies α at $r = 0.60$ while retaining attention, Joern-gated eligibility, and hard protection; $\alpha = 0$ removes only the soft structural prior. The resolved count increases from 20 at $\alpha = 0$ to 29 at $\alpha = 0.50$, then decreases to 23 at $\alpha = 1.00$. The nine-task range shows substantial variation, but larger structural weights do not consistently improve resolution. $\alpha = 0.25$ resolves one fewer task than $\alpha = 0.50$, with lower peak KV cache memory and end-to-end latency.

Controlled soft structural weight. Removing hard protection and disabling the small-candidate bypass yields smaller differences across α. With $r = 0.60$ fixed, Figure 5(c) shows resolved counts between 22 and 25, compared with 20–29 when hard protection is enabled. Increasing α from 0 to 1.00 changes the count from 23 to 25, but the intermediate values do not follow a monotonic trend. These results provide limited evidence that increasing the soft structural weight alone improves task resolution. Since this comparison changes both hard protection and the small-candidate bypass, the narrower range cannot be attributed to either change individually.

Structural components. Table 2 compares the structural components at $r = 0.60$, with all variants reporting 39.99% compression of eligible KV cache. Full CodeComp resolves 28 tasks, compared



(a) Retention-ratio ablation (b) Full-method α ablation (c) Controlled α ablation

Figure 5: End-to-end ablations on the same 50-task subset. (a) Varying the retention ratio r for full CodeComp with $\alpha = 0.25$. (b) Varying the soft structural-prior weight α at $r = 0.60$ while retaining attention-based selection, Joern-gated eligibility, and hard protection. (c) A controlled α ablation at $r = 0.60$ with hard protection and the small-candidate bypass disabled, isolating the effect of the soft structural score more closely. Each point reports the number of resolved tasks out of 50; empty outputs are counted as unresolved. Detailed measurements are reported in Appendix A.

with 24 without structural guidance and 25 with either gated attention alone or the soft prior without hard protection. Hard protection alone resolves 20 tasks, indicating that prioritizing protected code does not consistently improve task resolution. The combination achieves the highest resolved count in this comparison, although the individual components do not provide uniform gains. This improvement also involves an efficiency trade-off: full CodeComp uses 5.34 GiB of peak KV cache memory and takes 296.46 s, compared with 5.12 GiB and 252.81 s for the soft-only variant. The common eligible compression rate does not imply identical trajectories or overall memory use; overall KV cache compression ranges from 13.18% to 15.69% across the variants.

5 Conclusions

We presented CodeComp, a training-free KV cache compression method for coding agents that combines attention with source structure to compress historical READ KV cache while retaining the remaining context. On 100 SWE-bench Verified tasks with two coding models, CodeComp reduces peak KV cache memory while achieving resolve rates close to the uncompressed agent, although latency changes differ across models. Structural analysis shows improved retention across most examined code categories, while ablations indicate sensitivity to the structural weight and limited gains from individual components. These results motivate further study across coding tasks, programming languages, model families, and agent workflows.

Limitations

Our evaluation is limited to 100 SWE-bench Verified tasks per model, two coding models, and one agent scaffold; the ablations and structural retention analysis use 50 tasks. Single runs and small differences in resolved counts also leave uncertainty about the stability of method rankings and structural-weight effects. CodeComp depends on Joern annotations and accurate source-to-token alignment: unsupported or unaligned results remain uncompressed, uncached analyses add overhead, and static annotations may omit task-relevant dependencies. Finally, CodeComp compresses the KV cache only after full-prompt prefill, so it does not reduce prompt length, prefill computation, or context-window requirements.

Reproducibility statement

Reproducibility details are provided in the method description (§3), experimental setup (§4.1), and full ablation results in Appendix A. The anonymous code repository linked in the abstract contains the implementation and supporting configurations used in our evaluation.

AI USE STATEMENT

Generative AI tools were used solely for grammar correction and language polishing. All AI-assisted edits were reviewed by the authors, who take full responsibility for the final content of this paper.

REFERENCES

Reyna Abhyankar, Zijian He, Vikranth Srivatsa, Hao Zhang, and Yiying Zhang. INFERCEPT: Efficient intercept support for augmented LLM inference. In *Proceedings of the 41st International Conference on Machine Learning*, 2024. URL https://arxiv.org/abs/2402.01869.

Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant J. Nair, Ilya Soloveychik, and Purushotham Kamath. Keyformer: KV cache reduction through key tokens selection for efficient generative inference, 2024. URL https://arxiv.org/abs/2403.09054.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Baobao Chang, Junjie Hu, and Wen Xiao. PyramidKV: Dynamic KV cache compression based on pyramidal information funneling, 2024. URL https://arxiv.org/abs/2406.02069.

CheetahClaws Team. CheetahClaws: Agent harness infrastructure for long-horizon, multi-model, and tool-using AI systems, 2026. URL https://github.com/SafeRL-Lab/cheetahclaws.

Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. Locagent: Graph-guided llm agents for code localization, 2025. URL https://arxiv.org/abs/2503.09089.

Yuan Feng, Junlin Lv, Yukun Cao, Xike Xie, and S. Kevin Zhou. Ada-KV: Optimizing KV cache eviction by adaptive budget allocation for efficient LLM inference, 2024. URL https://arxiv.org/abs/2407.11550.

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, 2023. URL https://arxiv.org/abs/2310.01801.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*, 2021. URL https://arxiv.org/abs/2009.08366.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, 2026. URL https://doi.org/10.5281/zenodo.20953922.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems*, 37:1270–1303, 2024.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models, 2023. URL https://arxiv.org/abs/2310.05736.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression, 2024. URL https://arxiv.org/abs/2310.06839.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Joern Developers. Joern: Code analysis platform, 2026. https://github.com/joernio/joern.

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, 2024. URL https://arxiv.org/abs/2404.14469.

Xiang Liu, Zhenheng Tang, Peijie Dong, Zeyu Li, Yue Liu, Bo Li, Xuming Hu, and Xiaowen Chu. Chunkkv: Semantic-preserving kv cache compression for efficient long-context llm inference, 2025. URL https://arxiv.org/abs/2502.00299.

Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. CacheGen: KV cache compression and streaming for fast large language model serving, 2023a. URL https://arxiv.org/abs/2310.07240.

Zichang Liu, Aditya Desai, Fangshuo Liao, Weitao Wang, Victor Xie, Zhaozhuo Xu, Anastasios Kyrillidis, and Anshumali Shrivastava. Scissorhands: Exploiting the persistence of importance hypothesis for LLM KV cache compression at test time, 2023b. URL https://arxiv.org/abs/2305.17118.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024.

Mistral AI. Devstral-small-2-24b-instruct-2512. https://huggingface.co/mistralai/Devstral-Small-2-24B-Instruct-2512, 2025.

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, 2024. URL https://arxiv.org/abs/2410.14684.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems, 2023. URL https://arxiv.org/abs/2310.08560.

Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27B dense model, April 2026. URL https://qwen.ai/blog?id=qwen3.6-27b.

SWE-bench Team. SWE-bench Verified. https://huggingface.co/datasets/SWE-bench/SWE-bench_Verified, 2024.

Hanlin Tang, Yang Lin, Jing Lin, Qingsen Han, Shikuan Hong, Yiwu Yao, and Gongyi Wang. RazorAttention: Efficient KV cache compression through retrieval heads, 2024a. URL https://arxiv.org/abs/2407.15891.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context LLM inference, 2024b. URL https://arxiv.org/abs/2406.10774.

Xin Wang, Yasheng Wang, Yao Wan, Jiawei Wang, Pingyi Zhou, Li Li, Hao Wu, and Jin Liu. Code-mvp: Learning to represent source code from multiple views with contrastive pre-training, 2022. URL https://arxiv.org/abs/2205.02029.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021. URL https://arxiv.org/abs/2109.00859.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents, 2024. URL https://arxiv.org/abs/2407.01489.

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, 2024. URL https://arxiv.org/abs/2410.10819.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pp. 590–604, 2014. URL https://doi.org/10.1109/SP.2014.44.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement, 2024. URL https://arxiv.org/abs/2404.05427.

Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, Zhangyang Wang, and Beidi Chen. H_2o: Heavy-hitter oracle for efficient generative inference of large language models, 2023. URL https://arxiv.org/abs/2306.14048.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

A Appendix

We provide the complete measurements for the ablations in Figure 5. All experiments use Qwen3.6-27B on the same 50 SWE-bench Verified tasks, with one run per configuration. Resolved counts include all submitted tasks; empty outputs are counted as unresolved. Prompt-token counts are accumulated across model calls and averaged over tasks. Eligible KV cache compression is measured over the compression pool, whereas overall compression includes the full prompt. Each configuration is evaluated through complete agent runs, so subsequent tool calls, prompt lengths, and absolute KV cache requirements can differ.

A.1 Retention Ratio

Table 3 examines the effect of retention ratio at $\alpha = 0.25$. Reducing r increases overall KV cache compression, from 10.63% at $r = 0.80$ to 26.99% at $r = 0.20$. The lowest peak memory is 4.61 GiB at $r = 0.40$, while the highest resolved count is 28 at $r = 0.60$, with 5.34 GiB of peak memory. Both stronger and weaker compression resolve fewer tasks on this subset: the other four ratios yield 24–25 resolved tasks. Retaining more tokens therefore does not consistently improve task resolution.

The memory reduction does not translate directly into lower end-to-end latency. The $r = 0.20$ and $r = 0.40$ runs take approximately 303 s, compared with 296.46 s at $r = 0.60$ and 282.51 s without eviction at $r = 1.00$. Cumulative prompt usage also differs across these runs, ranging from 153,147 to 189,052 tokens per task. End-to-end latency includes all model calls and tool execution, so these comparisons reflect changes in both inference costs and agent trajectories.

Table 3: Retention-ratio ablation on 50 SWE-bench Verified tasks using Qwen3.6-27B, with $\alpha = 0.25$.

Method	Resolved (/50)	Empty Outputs	Prompt Tokens / Task	Eligible KV Cache Compression (%)	Overall KV Cache Compression (%)	Peak KV Cache Memory (GiB)	Agent E2E (s)
CodeComp $r = 0.20$	25	7	189,052	79.99%	26.99%	4.67	303.04
CodeComp $r = 0.40$	25	3	179,689	59.99%	21.27%	4.61	303.15
CodeComp $r = 0.60$	28	3	184,874	39.99%	14.18%	5.34	296.46
CodeComp $r = 0.80$	24	3	184,513	19.99%	10.63%	5.32	340.06
CodeComp $r = 1.00$	25	1	153,147	0.00%	0.00%	5.71	282.51

A.2 Soft Structural Guidance

Table 4 varies the soft structural weight at $r = 0.60$, with attention, Joern-gated eligibility, and hard protection enabled throughout. Setting $\alpha = 0$ removes only the soft prior. All configurations report 39.99% eligible KV cache compression, while the resolved count ranges from 20 to 29. Resolution

peaks at $\alpha = 0.50$ and then declines as the weight increases, indicating sensitivity to the balance between attention and structural scores.

The highest resolved count does not coincide with the lowest memory or latency. Compared with $\alpha = 0.50$, $\alpha = 0.25$ resolves one fewer task but reduces peak KV cache memory from 5.53 to 5.34 GiB and latency from 312.77 to 296.46 s. Increasing the weight to $\alpha = 1.00$ yields the lowest latency in this comparison, 293.10 s, but resolves only 23 tasks and uses the most peak memory at 5.99 GiB. The shared eligible compression rate therefore does not imply identical memory use or execution time across the resulting agent trajectories.

Table 4: Soft structural-weight ablation on 50 SWE-bench Verified tasks using Qwen3.6-27B at $r = 0.60$. Attention-based ranking, Joern-gated eligibility, and hard protection remain enabled; $\alpha = 0$ removes only the soft prior.

Method	Resolved (/50)	Empty Outputs	Prompt Tokens / Task	Eligible KV Cache Compression (%)	Overall KV Cache Compression (%)	Peak KV Cache Memory (GiB)	Agent E2E (s)
CodeComp $\alpha = 0.00$	20	6	139,612	39.99%	16.52%	5.40	298.76
CodeComp $\alpha = 0.25$	28	3	184,874	39.99%	14.18%	5.34	296.46
CodeComp $\alpha = 0.50$	29	4	161,135	39.99%	16.57%	5.53	312.77
CodeComp $\alpha = 0.75$	24	4	159,585	39.99%	14.79%	5.48	293.36
CodeComp $\alpha = 1.00$	23	5	150,338	39.99%	16.70%	5.99	293.10

A.3 Controlled Soft Structural Guidance

Figure 6 examines the soft prior with hard protection and the small-candidate bypass disabled at $r = 0.60$. The resolved count varies from 22 to 25, a narrower range than in the preceding ablation. Increasing α from 0 to 1.00 raises the count from 23 to 25, but the intermediate weights do not show a monotonic improvement. Equal resolved counts can also accompany different numbers of empty outputs: $\alpha = 0.50$ and $\alpha = 0.75$ both resolve 24 tasks, with nine and four empty outputs, respectively. Reporting both measures distinguishes task resolution from whether the agent produces an output.

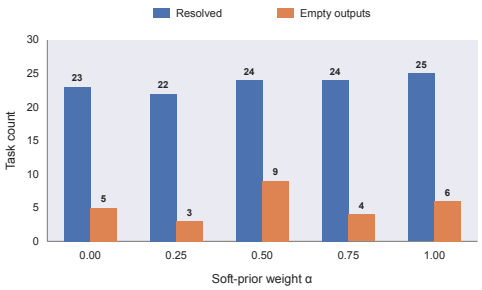


Figure 6: Controlled soft-prior ablation on 50 tasks at $r = 0.60$.

This comparison provides limited evidence of a consistent improvement from increasing the soft weight alone. It also disables two mechanisms relative to the preceding ablation, so the smaller variation cannot be attributed specifically to removing hard protection or the small-candidate bypass.

A.4 Related Work

KV cache compression. Existing methods reduce KV cache costs through token selection, budget allocation, and cache representation. Scissorhands, FastGen, and Keyformer retain tokens estimated to be useful for subsequent generation (Liu et al., 2023b; Ge et al., 2023; Adnan et al., 2024). PyramidKV and Ada-KV adapt retention budgets across layers or heads (Cai et al., 2024; Feng et al., 2024), while DuoAttention and RazorAttention distinguish heads associated with long-range retrieval from those serving local attention (Xiao et al., 2024; Tang et al., 2024a). Other approaches address cache access and storage: Quest selects pages using the current query (Tang et al., 2024b), and CacheGen compresses cached representations for transmission (Liu et al., 2023a). CodeComp focuses on the content being selected in coding-agent prompts. It restricts eviction to eligible historical READ results and combines attention with source-level structural information.

Context management for interactive LLMs. Interactive inference requires managing context across repeated model calls and tool execution. INFERCEPT addresses interruptions through request scheduling and KV cache swapping (Abhyankar et al., 2024). LLMLingua and LongLLMLingua reduce input length by compressing prompt content (Jiang et al., 2023; 2024), while MemGPT manages information across in-context and external memory (Packer et al., 2023). These approaches

operate at different stages of context processing. CodeComp retains the complete serialized prompt for prefill and compresses the KV cache used during decoding. It can therefore be applied alongside context summarization, whose combination with CodeComp is evaluated in §4.

Structure-aware code modeling. Program analysis has informed both model inputs and training objectives for code. GraphCodeBERT incorporates data-flow information into pre-training (Guo et al., 2021), CodeT5 uses identifier-aware objectives (Wang et al., 2021), and CODE-MVP combines multiple views of source code, including syntax and control flow (Wang et al., 2022). These methods incorporate code-specific information when learning representations. CodeComp uses structural annotations at inference time to assign retention priorities and supplement attention scores. It requires no changes to model parameters or additional training.

Structure-aware repository agents. Repository agents use program structure to locate relevant code and support navigation. AutoCodeRover searches classes and methods using an abstract syntax tree (Zhang et al., 2024). RepoGraph supplies graph-based repository navigation (Ouyang et al., 2024), and LocAgent uses a heterogeneous code graph for multi-hop localization (Chen et al., 2025). CodeComp applies structural information after code has been retrieved, when selecting the KV cache retained for decoding. It preserves the agent's tool interface and stored messages, although compression can affect subsequent generation and tool calls. Its use of program structure therefore concerns inference memory rather than the retrieval procedure itself.