BEYOND TOKEN SAVINGS: A SYSTEMATIC STUDY OF CONTEXT COMPRESSION IN LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

As LLM agents tackle longer tasks, they increasingly compress growing histories of reasoning, actions, and tool outputs. Compression can reduce token use, but it also changes the information available for later decisions. Existing agentic harnesses bundle decisions about what to compress, when to compress, and how much to remove into fixed policies. A systematic characterization is needed to disentangle these decisions and reveal how each affects task success and execution cost. We systematically vary these decisions across three open-weight models on SWE-bench Verified and Terminal-Bench 1.0. Across nearly 35,000 agent runs, we measure task success, token use, end-to-end latency, and estimated cost. We find that fewer tokens need not mean faster or cheaper execution: on Terminal-Bench with Qwen, policies using roughly one-third as many tokens can take 20–80% longer than the uncompressed agent. Policies with similar overall success can solve different tasks, while the same policy can perform quite differently across models. Our results motivate evaluating compression by its effects on agent execution and tailoring policies to the task, model, and workload.

1 INTRODUCTION

Agentic systems increasingly execute long-horizon workflows that interleave LLM reasoning, tool calls, code execution, and environment interaction. As these trajectories grow to hundreds of steps (Figure 1), their accumulated history can approach or exceed finite context windows (Hu et al., 2025; Packer et al., 2023). Even before reaching these limits, processing increasingly long contexts raises execution cost and can make relevant information harder for models to use effectively (Tian et al., 2025; Du et al., 2025; Liu et al., 2024; Levy et al., 2024). So, long-running agents need mechanisms for controlling how much of their history is carried forward.

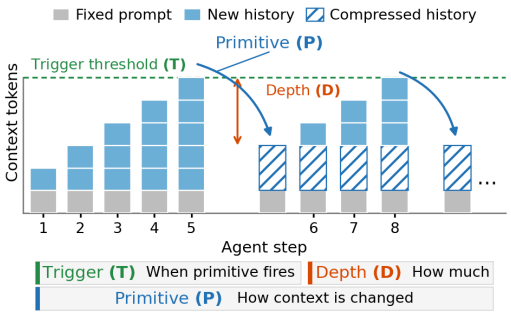


Figure 1: Agent context grows across steps and is repeatedly compressed by a policy defined by primitive (P), trigger (T), and depth (D).

Context compression has emerged as one such mechanism. Modern coding agents selectively remove, rewrite, or summarize portions of an execution trajectory as it grows (Anthropic, 2026b; OpenAI, 2026b). Its impact is already substantial in production: in a trace of 13 million GitHub Copilot sessions, sessions requiring compaction account for 44.2% of all tokens served, and the median compaction event removes 72.8% of the existing context (Liu et al., 2026). Compression, however, is not simply a mechanism for saving tokens or fitting within a context window. It changes the information available to subsequent agent decision. Removing information that becomes useful later can alter the agent's trajectory and cause a task to fail that it would otherwise have solved (Liu et al., 2026); compression mechanisms that invoke an LLM can themselves add token and latency overheads.

A context-compression system must decide *how* to compress context, *when* to trigger compression, and *how much* context to remove. Existing systems make different choices along each dimension. Some discard older interactions (OpenAI, 2025; LangChain, 2025), while others summarize them

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.

using additional LLM calls (OpenAI, 2026b; Anthropic, 2026b). They also differ in when compression is triggered and how much history is retained (Packer et al., 2023; Wang et al., 2025). These choices cannot be considered independently. Compressing early may reduce the context processed by later steps, but it may also cause more compression events. Removing more context can further reduce token usage, but also increases the chance of discarding information needed later.

Because these choices interact, end-to-end evaluations of complete policies are difficult to interpret. A policy may improve task success because of its compression mechanism, its trigger point, the amount of context it retains, or an interaction among these choices. Similarly, two policies with nearly identical aggregate resolve rates may alter the trajectories of entirely different tasks. So, evaluating complete policies leaves a basic question unanswered: *how do the individual decisions that constitute a compression policy shape task success, execution cost, and one another?* Moreover, because models differ in how they use long and compressed contexts, it is unclear whether conclusions obtained for one model transfer to another.

In this work, we systematically characterize context compression as a runtime policy that jointly controls execution cost and the information available to future agent decisions. We vary the compression mechanism, trigger point, and aggressiveness, and organize our study around three questions.

1. What tradeoffs do different compression policies create between task success and execution cost? Compression can reduce the context processed during execution, but the resulting savings depend on when compression occurs, how much context is removed, and the cost of compression itself. At the same time, removing useful information can reduce task success. We characterize how these policy choices jointly affect resolve rate, token consumption, and end-to-end latency.

2. How does compression change which tasks an agent solves? Aggregate resolve rate can hide substantial changes in task-level behavior. Two policies may achieve similar overall success rates while solving different tasks: compression can turn some failures into successes while causing previously successful tasks to fail. So, we compare outcomes at the task level to reveal effects that aggregate metrics obscure.

3. Do the effects of compression policies transfer across models? A compression policy changes the information presented to the model, so its effect may depend on how that model uses long and compressed contexts. We study whether policies exhibit similar task-success and efficiency tradeoffs across models, or whether effective choices must be tuned to the model executing the task.

To answer these questions, we conduct a controlled, large-scale evaluation of context-compression policies. We examine three policy dimensions, compression mechanism, trigger point, and aggressiveness, while holding the agent loop and task set fixed. We evaluate 65 policies across three open-weight models on SWE-bench Verified and TerminalBench, and repeat each policy–model–task combination three times to quantify run-to-run variability, yielding nearly 35,000 agent executions.

Our study reveals that context compression is not a single efficiency knob: its mechanism, timing, and aggressiveness can change both execution cost and the tasks an agent ultimately solves. We further find that conclusions drawn for one model do not necessarily carry over to another. We quantify these effects in detail in §4 and use them to derive implications for the design and evaluation of context-compression policies.

2 Design Space of Compression Policies

Context compression is already used by widely deployed agents from OpenAI, Anthropic, and Google (Google, 2026a; Microsoft, 2026; OpenAI, 2026a; Anthropic, 2025), as well as open-source systems (OpenHands, 2026; Cline, 2026; Pi Coding Agent, 2026). These systems differ in their implementations, but each must make three fundamental decisions: what information to remove or rewrite, when to compress, and how much context to remove. Existing systems typically bundle these decisions into particular compression policies, making it difficult to understand which choice is responsible for an observed change in task success or execution cost. We decompose a compression policy along three axes: *primitive*, *trigger*, and *depth*. The primitive determines *how* the trajectory is compressed; the trigger determines *when* compression occurs; and the depth determines *how much* context is removed. Together, one choice along each axis defines a compression policy. This decomposition lets us vary the three decisions independently and isolate their effects on task success, token consumption, and latency. It also provides a common design space for comparing compression strategies used by existing agents. We describe each axis next.

2.1 COMPRESSION PRIMITIVES

The primitive determines how information is removed or rewritten during a compression event. Different primitives preserve different parts of the trajectory. For example, truncation removes older agent steps while preserving recent history, tool-result clearing removes older tool outputs while retaining the surrounding interaction, and summarization replaces prior history with a shorter LLM-generated representation. Thus, the choice of primitive determines what evidence, actions, and intermediate state remain available to the agent after compression.

We group primitives into three families according to how they remove information: rule-based primitives delete or clear context with a fixed procedure, LLM-based primitives rewrite context into a shorter representation, and stacked primitives apply both in sequence. Across these families, we study eight primitives drawn from deployed agentic systems(Table 2). Every primitive operates only on the *compressible window*, which is the accumulated assistant messages and tool results from the trajectory. The system prompt and task instruction are always kept verbatim. Each primitive is defined only by what it removes from the compressible window. The trigger determines when the primitive is applied (§2.2), while the depth determines how much context it removes (§2.3).

Rule-based primitives. Rule-based primitives remove information using a fixed procedure and require no additional model invocation. We study two forms used by existing systems.

- *Truncation (TR)* removes the oldest messages from the compressible window until the target context size is reached (OpenAI, 2025; LangChain, 2025). It preserves the most recent trajectory verbatim, but discards earlier interactions entirely.
- *Tool-result clearing (TRC)* replaces older tool outputs with a short placeholder such as `[output cleared]`, while retaining the surrounding agent interactions and recent tool results (Anthropic, 2025). Unlike truncation, TRC selectively removes a particular source of context growth rather than dropping an entire prefix of the trajectory.

These primitives illustrate an important distinction. TR decides what to preserve primarily by *recency*, whereas TRC decides by *content type*.

LLM-based primitives. LLM-based primitives instead preserve prior context by rewriting it into a shorter representation. A summarizer receives some portion of the compressible window and produces a summary that replaces the original text. Unlike rule-based deletion, summarization can retain information from across the compressed history, but it requires an additional model invocation and can omit or alter information during rewriting. The LLM-based primitives we study vary in two ways: *which portion of the trajectory is summarized* and *how the summary is structured*. A primitive may summarize the entire compressible window or only its older portion, leaving recent interactions verbatim. The generated summary may also be free-form or constrained to preserve predefined categories of information. Unless otherwise stated, we use the same model for agent execution and summarization; we separately evaluate sensitivity to the summarizer model in §A.6.3.

Stacked primitives. Finally, stacked primitives combine rule-based removal with LLM-based summarization. For example, a policy can first clear older tool results and then summarize the remaining history. This mirrors deployed systems that combine multiple forms of context reduction rather than relying on a single operation (Google, 2026a; Anthropic, 2026a). Stacking can reduce the amount of text passed to the summarizer, but also changes which information is available when the summary is produced. Thus, these primitive families differ not only in how much context they remove, but in *which information survives compression and in what form*: verbatim recent history, selected content types, or a rewritten representation of prior interactions.

2.2 COMPRESSION TRIGGERS

The trigger determines when compression occurs, shaping how much history accumulates and how often the agent intervenes to reduce it. We study two strategies: *threshold-based* triggers tied to context size and *step-based* triggers tied to agent steps.

Threshold trigger (Th). Compression fires when the trajectory exceeds a token threshold and can recur as context grows afterward. Lower thresholds trigger earlier and potentially more often; higher thresholds allow more history to accumulate. The threshold value is a trigger parameter, not a separate policy axis. We evaluate tight, primary, and loose settings for Qwen and Devstral, and only the primary setting for GLM (§3), to study sensitivity to when compression occurs.

Step trigger (St). A step trigger instead invokes compression as the agent executes, independent of the current context size. In our evaluation, compression runs after every agent step. Thus, the

number of compression events is determined by trajectory length rather than by how quickly the trajectory accumulates tokens. We evaluate the step trigger with tool-result clearing, which allows old tool outputs to be removed continuously as they accumulate.

These two triggers impose different notions of *when context has become worth compressing*: threshold triggers respond to context size, whereas step triggers respond to trajectory progress.

2.3 Compression Depth

Once compression is triggered, the depth determines how much of the current context is removed. This creates another tradeoff. Shallow compression preserves more of the existing trajectory, but leaves less room for subsequent context growth and may cause compression to fire again soon. Deeper compression creates more headroom, but removes more information at each event. We define compression depth, D, as $D = 1 - \frac{\texttt{target tokens}}{\texttt{current tokens}}$, where `current tokens` is the size of the compressible window when compression begins and `target tokens` is its target size after compression. Thus, $D = 0.5$ reduces the compressible window to half its pre-compression size, while larger values indicate more aggressive compression. We evaluate $D \in \{0.3, 0.5, 0.7\}$, with $D = 0.5$ as the canonical setting. We distinguish *depth-tunable* primitives, such as truncation and summarization, whose retained history or summary size can be targeted, from *depth-invariant* primitives, such as tool-result clearing, whose effective depth depends on the operation and available content. We sweep depth only for depth-tunable primitives.

2.4 Compression Policies

A compression policy combines a primitive, a trigger, and, when applicable, a depth. We denote a policy as $P/T/D$, where P identifies the primitive, T the trigger, and D the compression depth. For example, $SU\text{-}p/Th/0.5$ denotes partial summarization triggered by a token threshold and targeting a 50% reduction in the compressible window. Appendix A.2.2 maps representative deployed systems onto this design space.

3 Characterization Methodology

We use the design space from §2 to systematically characterize how compression primitives, triggers, and depths affect agent task success and execution cost. Our methodology is designed to separate the effects of these policy choices while controlling the agent harness, task, and model within each comparison. We evaluate policies across two agentic coding benchmarks and three open-weight models, and repeat each configuration to account for run-to-run variability.

Benchmarks and Tasks. We evaluate compression policies on two coding benchmarks, SWE-bench Verified (Jimenez et al., 2024) and Terminal-Bench 1.0 (Merrill et al., 2026). In SWE-bench Verified, the agent resolves a real GitHub issue by modifying a repository, and held-out tests evaluate the resulting patch. In Terminal-Bench, the agent completes command-line tasks through multi-step interaction with a terminal. For SWE-bench, we follow prior work (Wang et al., 2025; Hou et al., 2024) and use 100 issues from Django, SymPy, and scikit-learn. For Terminal-Bench, we use 40 tasks. The threshold and depth sweeps use subsets of 25 and 15 of these tasks, respectively. Appendix A.3 lists the cohorts and their composition.

Agent harness. We use mini-swe-agent (Yang et al., 2024), whose minimal agent loop adds no memory or context-management mechanism that could confound compression, and we apply benchmark-specific step and time limits (Appendix A.3).

Models. We evaluate Qwen3.5-35B-A3B, Devstral-Small-2-24B, and GLM-4.7-Flash to test whether compression effects transfer across models. Models are served locally using vLLM (Kwon et al., 2023), with the agent model also performing summarization by default. Appendix A.3 provides model and serving details.

Trigger-Threshold Calibration. Threshold-based policies require choosing the context size at which compression is triggered. Setting the threshold only at the model's maximum context window would evaluate compression primarily as a response to imminent context overflow. Our goal is broader: to characterize when compression becomes useful during normal agent execution, including trajectories that never approach the model's context limit.

So, we calibrate trigger thresholds from the context demand of uncompressed trajectories. For each model and benchmark, we first execute the tasks without compression and record the peak per-call context size reached by each trajectory. We then define three trigger thresholds settings from

		Depth-tunable					Depth-invariant						Reference	
Trigger		TR	SU	SU-p	SS	SS-p	TRC	TRC+SU	TRC+SS	OTRC+TR	OTRC+SU-p	OTRC+SS-p	FC	OTRC
Threshold T		Th	Th	Th	Th	Th	Th	Th	Th	$St{+}Th$	$St{+}Th$	$St{+}Th$	–	St
	Tight	○ ○ ○	○ ○ ○	○ ○ ○	○ ○ ○	○ ○ ○	○	○	○	○	○	○		
Threshold	Primary	○ ● ○	○ ● ○	○ ● ○	○ ● ○	○ ● ○	●	●	●	●	●	●	●	●
	Loose	○ ○ ○	○ ○ ○	○ ○ ○	○ ○ ○	○ ○ ○	○	○	○	○	○	○		

Legend: ● Full comparison, ○ Controlled sweeps (task sets defined in §3).

Table 1: Policy grid, run for each model at its three threshold settings from §3. In each cell the marks read left to right as $D = 0.3, 0.5, 0.7$. Depth-invariant primitives do not depend on D and therefore have a single mark per row. FC and OTRC has no threshold, so each is one cell. Every cell is run three times per task.

this distribution: *tight*, *primary*, and *loose*, corresponding to its 5th, 15th, and 25th percentiles, respectively. Lower thresholds expose more trajectories to compression and tend to trigger it earlier, while higher thresholds restrict compression to trajectories with larger context demands.

This calibration serves two purposes. First, it places the trigger relative to the context demands of each model and workload rather than using the same arbitrary token count across models with different context behavior. Second, the three threshold settings let us study whether conclusions about a compression policy depend on when it is invoked. We determine all thresholds from uncompressed runs before evaluating the corresponding compressed configurations. Appendix A.4 reports the resulting thresholds and context-demand distributions.

Characterization Matrix. We organize our experiments into two complementary tracks (Table 1). The first provides a broad comparison across compression policies, while the second varies individual policy parameters to isolate their effects.

Full comparison. We evaluate all compression primitives on the full benchmark task sets using each model's primary threshold and the canonical compression depth $D = 0.5$. This comparison captures how substantially different compression strategies affect task success, token consumption, and latency under a common operating point. We also include the full-context reference and the step-triggered tool-result-clearing policy.

Controlled sweeps. We use the smaller stratified task subsets to vary trigger threshold and compression depth. For depth-tunable primitives, we evaluate $D \in \{0.3, 0.5, 0.7\}$ at each of the tight, primary, and loose thresholds. Primitives whose effective depth is determined by the operation itself are evaluated once per threshold. These sweeps let us separate the effect of *when* compression occurs from *how aggressively* it removes context without requiring the full combinatorial grid on every task.

Every task–policy–model configuration is repeated three times to capture run-to-run variability. Table 1 summarizes the resulting evaluation matrix; Appendix A.3 (Table 4) reports the complete configuration and run counts for both benchmarks.

Metrics and analysis. We measure task success, token consumption, billed input cost, and end-to-end latency over three runs per task–policy–model configuration, examining both aggregate and task-level outcomes. Appendix A.5 details metric definitions, inclusion rules, and statistical comparisons; Appendix A.6.4 reports run outcomes and resource limits.

4 Results

Question #1 (Q1). What tradeoffs do different compression policies create between task success and execution cost?

Figure 2 shows Qwen3.5-35B-A3B on SWE-bench (top) and Terminal-Bench (bottom). Resource metrics are ratios of task-averaged values to Full Context (FC) on shared eligible tasks, denoted by "/ FC"; 0.5 means half the FC value. The x-axis shows relative token usage; the columns show resolve-rate change in percentage points over all runs, relative latency, and relative billed cost (Appendix A.5.2), respectively.

Observation #1: Context compression consistently reduces token usage, but can trade off success, latency, and billed cost in different ways.

Tradeoffs observed on SWE-bench. Figure 2 shows that every compression policy reduces token usage relative to full context. On SWE-bench, policies use roughly 0.4–0.7× the tokens, while

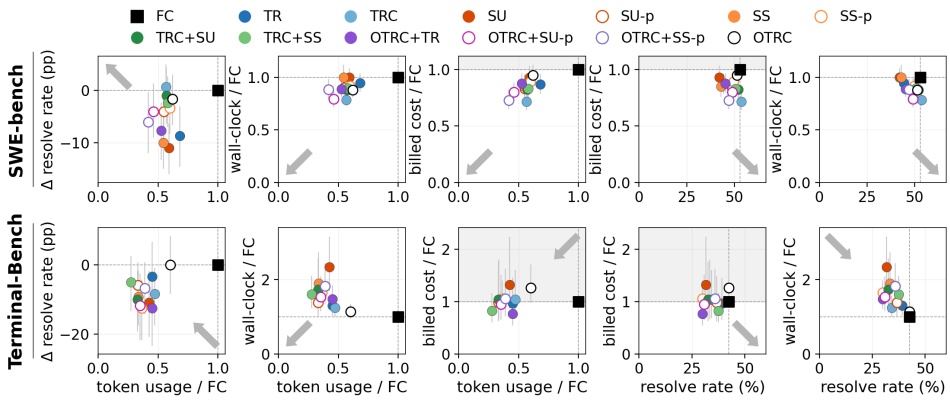


Figure 2: Qwen's success and resource tradeoffs on SWE-bench (top) and Terminal-Bench (bottom). Resource metrics are paired ratios to full context on completed runs; success includes all runs. Gray shading marks higher billed cost, and dotted diagonals mark equal proportional token and bill savings. Vertical bars show 95% task-bootstrap intervals.

success ranges from 11.0 percentage points (pp) below full context to 0.7 points above it. TRC mostly matches full-context success while reducing token usage, latency, and billed cost to $0.57\times$, $0.79\times$, and $0.71\times$, respectively. SU uses a similar $0.59\times$ the tokens but reduces success by 11.0 pp.

Similar token usage, different outcomes. These differences persist across policies with comparable token consumption. On SWE-bench, policies using roughly 0.56–$0.63\times$ the tokens of full context span about 12 percentage points in resolve rate, 0.79–$1.0\times$ in latency, and 0.71–$0.95\times$ in billed cost. Even similar success rates can mask different execution costs: TRC and OTRC achieve resolve rates of 53.7% and 51.3%, respectively, yet their billed costs are $0.71\times$ and $0.95\times$ full context, and their latencies are $0.79\times$ and $0.88\times$.

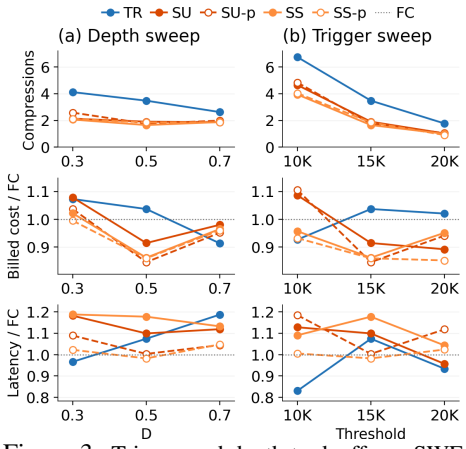
Trade-offs observed on Terminal-Bench. On Terminal-Bench, most policies reduce token usage more aggressively, to roughly 0.25–$0.4\times$ full context, but 11 of the 12 reduce success by 5–13 percentage points. All policies also take longer than full context, with latency ranging from roughly $1.05\times$ to $1.8\times$ for most policies. Billed cost nevertheless decreases for 9 of the 12 policies, with several reaching roughly 0.4–$0.7\times$ full context. We suspect that the workloads contribute to this difference: SWE-bench agents can recover working context from code and patches, whereas Terminal-Bench's varied workflows may depend more on intermediate observations and action history.

Token savings alone therefore do not identify an effective or efficient policy. The same reduction in tokens can accompany different success rates and execution costs, motivating a closer look at what determines billed cost and latency.

Observation #2: Token savings do not yield proportional billed-cost or latency savings.
Observed mismatch. The middle and right columns of Figure 2 show how large this mismatch can be. On SWE-bench, using roughly $0.6\times$ the tokens of full context still incurs 0.71–$0.95\times$ the billed cost and 0.79–$1.0\times$ the latency. On Terminal-Bench, policies using roughly one-third of the tokens can take 1.2–$1.8\times$ as long, and some even cost more than full context. Compression can therefore save tokens without delivering comparable monetary savings, and can increase runtime despite much smaller prompts. Total token usage combines the tokens processed per call with the number of calls, but does not distinguish the factors that determine billed cost and latency.

Billed cost: fresh vs cached input. Billed cost depends on how many input tokens remain eligible for the cached rate. Under the assumed prefix-cached pricing, cached input tokens cost one-tenth as much as fresh input tokens (Appendix A.5.2). A policy that frequently rewrites the trajectory can remove tokens while invalidating much of the reusable prefix. It may therefore process fewer tokens but charge more of them at the fresh-input rate, producing a similar or even higher estimated bill.

Latency: call duration and execution length. Total latency depends on both the duration and number of calls, as well as tool execution and compression overhead. Shorter prompts can make individual model calls faster, yet additional agent calls, tool execution, and compression overhead can offset those savings. Step-triggered policies make the largest reductions in tokens per step but require 10–27% more calls across models. On Qwen, faster calls more than offset the additional calls;



Highest resolve Lowest cost Lowest latency

(c) Resolve, cost, and latency FC: 65.3% / 1.00× / 592s

Primitive	Trigger	Shallow (0.3) RR (%)	Shallow (0.3) Cost/FC	Shallow (0.3) Lat. (s)	Depth 0.5 RR (%)	Depth 0.5 Cost/FC	Depth 0.5 Lat. (s)	Deep (0.7) RR (%)	Deep (0.7) Cost/FC	Deep (0.7) Lat. (s)
TR	10K	**44.0**	1.12	555	41.3	0.90	**469**	37.3	**0.87**	586
	15K	**61.3**	1.05	**617**	46.7	1.03	640	50.7	**0.90**	669
	20K	54.7	1.23	640	**62.7**	0.99	**548**	60.0	**0.88**	718
SU	10K	46.7	**1.07**	**631**	**50.7**	1.10	653	48.0	1.11	645
	15K	50.7	1.07	678	53.3	0.98	652	**60.0**	**0.94**	**612**
	20K	57.3	1.00	674	**62.7**	**0.92**	**599**	56.0	0.99	689
SU-p	10K	**52.0**	1.13	709	46.7	**1.07**	668	49.3	1.09	**636**
	15K	**66.7**	1.08	660	60.0	**0.86**	**607**	64.0	0.97	639
	20K	64.0	0.90	615	**65.3**	0.94	636	62.7	**0.87**	**608**
SS	10K	**56.0**	0.92	**596**	52.0	0.94	622	44.0	**0.83**	820
	15K	**57.3**	1.04	657	54.7	**0.89**	796	56.0	0.96	**618**
	20K	62.7	1.00	665	58.7	0.99	677	**64.0**	**0.94**	**628**
SS-p	10K	**60.0**	0.98	627	50.7	0.93	**582**	56.0	**0.91**	789
	15K	**70.7**	0.95	577	60.0	**0.91**	677	60.0	0.92	**564**
	20K	65.3	0.94	627	**73.3**	**0.83**	591	68.0	0.86	**585**

Figure 3: Trigger and depth tradeoffs on SWE-bench (Qwen). Left: compression frequency, input cost, and latency. Right: resolve rate, cost relative to FC, and latency per setting; yellow, green, and circles mark best resolve, cost, and latency across depths. Left resource curves exclude time-capped attempts; the table includes all attempts.

on Devstral, calls themselves become slower, compounding the increase in execution length. In contrast, stacked threshold policies reduce token usage by 22–55% while keeping call counts near full context and latency at 0.88–$1.01\times$.

These results explain why token savings alone cannot predict execution cost: compression also changes which tokens receive the cached rate, how many calls the agent makes, and how much time each step takes.

Observation #3: Trigger threshold and depth shape different execution tradeoffs.
Experiment. To separate when compression occurs from how much history it removes, we vary the trigger threshold at fixed depth and compare depths at fixed thresholds. Figure 3 reports the Qwen SWE-bench sweeps on 25 tasks, with three attempts per task: thresholds of 10K, 15K, and 20K tokens and depths $D \in \{0.3, 0.5, 0.7\}$. We measure compression frequency, resolve rate, billed input cost, and end-to-end latency.

Threshold effects. Delaying compression generally helps in these sweeps. Increasing the threshold from 10K to 20K at $D = 0.5$ reduces average compression frequency from 4.85 to 1.15 events per attempt and improves resolve rate across all tested primitives. For SU, resolve rate rises from 50.7% to 62.7%, while input cost drops from $1.10\times$ to $0.92\times$ full context and latency falls from 653 to 599 seconds. Retaining more history before compression can therefore reduce repeated interventions while improving success and efficiency.

Depth effects. The effect of depth depends on the primitive. At a 15K threshold, increasing TR's depth from 0.3 to 0.7 lowers compression frequency and input cost, but resolve rate drops from 61.3% to 50.7% and latency increases. SU shows the opposite trend: resolve rate rises from 50.7% to 60.0% while cost and latency decrease. Excluding time-capped attempts preserves these directions: TR loses 15.4 percentage points of success, whereas SU gains 7.4 points. Across the 15 primitive–threshold combinations, shallow compression gives the highest resolve rate in eight cases; deep compression gives the lowest cost in eight and the lowest latency in seven.

Differences across benchmark suites. Although both benchmarks use the same calibration procedure, larger tool outputs on Terminal-Bench rebuild context more quickly, triggering compression more frequently than on SWE-bench. On shared solved tasks, Qwen with TR compresses about 9 times per run on Terminal-Bench, compared with fewer than 2 on SWE-bench. For five policies, total token usage remains at or above $0.94\times$ full context despite shorter prompts, while summarization calls account for 28–46% of solved-run wall-clock time for the 4 summarization primitives.

Takeaway 1. No single compression policy dominates across task success and execution cost. So, the choice of a policy needs to reflect the workload and the user's priorities for success, latency, and cost.

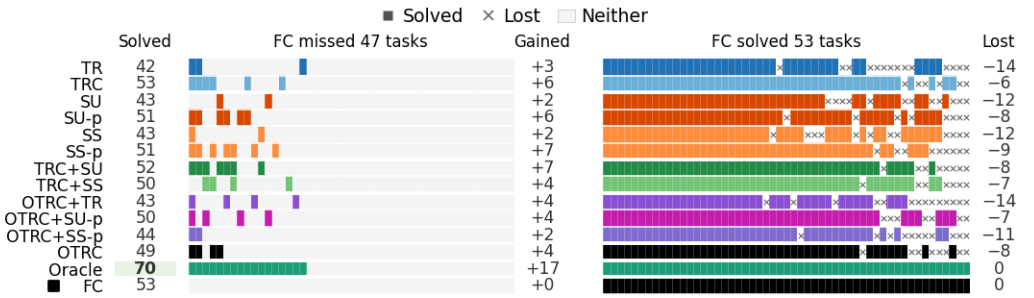


Figure 4: Qwen on SWE-bench. Task coverage relative to FC, with gains, losses, and total solved.

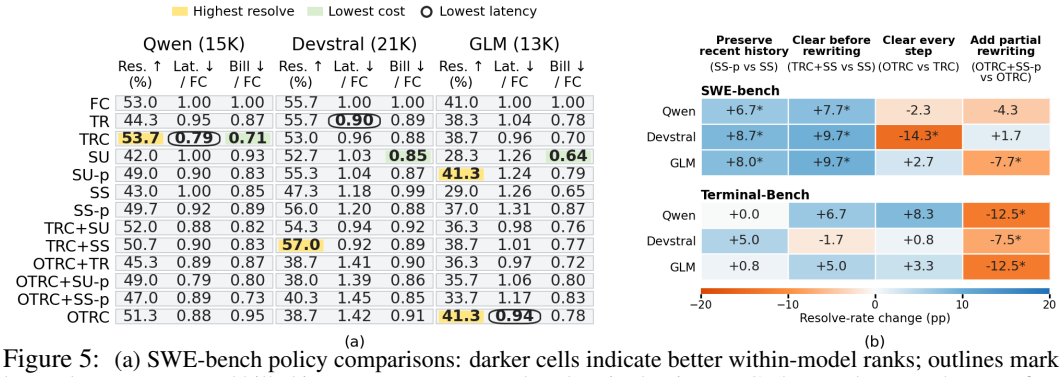


Figure 5: (a) SWE-bench policy comparisons: darker cells indicate better within-model ranks; outlines mark best values. Latency and billed input cost use uncapped, task-paired ratios to FC. (b) Resolve-rate changes (first policy minus second); * marks 95% task-bootstrap intervals excluding zero. Primary thresholds; three attempts per task (SWE-bench: 100 tasks; Terminal-Bench: 40).

Question #2 (Q2). How does compression change which task an agent can solve?

Q1 shows that some policies approximately match full-context success at lower execution cost, but may solve different tasks. We therefore compare tasks gained and lost relative to FC, counting a task as solved when at least two of three attempts succeed. Appendix A.6.2 shows the breakdown for Qwen's 100 tasks, separating the 47 that FC misses from the 53 it solves; Appendix A.5 details outcome handling.

Observation #4: Similar overall performance can mask substantial task-level differences.

Figure 4 shows substantial task turnover under compression. Across compression policies, 2–7 of the 47 tasks that full context misses become newly solved, while 6–14 of the 53 tasks solved by full context become unsolved. These gains and losses can largely offset each other. For example, TRC+SU solves 52 tasks compared with 53 for full context, but newly solves 7 tasks while losing 8 previously solved tasks. SS-p shows a similar pattern, gaining 7 tasks and losing 9, for a total of 51 solved tasks. In contrast, TR gains only 3 tasks while losing 14, reducing the total to 42. Thus, approximately matching full-context success does not mean preserving its task coverage: gains on other tasks can mask failures on tasks that full context previously solved.

Takeaway 2. The best compression policy varies by task, motivating task-aware customization rather than a single fixed policy.

Question #3 (Q3). Do compression policies effects transfer across models?

Q1 and Q2 show that compression changes both execution cost and which tasks an agent solves. We now examine whether these effects transfer across models. Figure 5 compares policy rankings for Qwen, Devstral, and GLM, then tests whether individual design choices have consistent effects despite differences in the best-performing policies.

Observation #5: Compression-policy effects can change substantially across models.

Same policy, different effects. Figure 5a shows that the same compression policy can have very different effects on resolve rate, latency, and billed cost depending on the model. For example,

OTRC performs well with Qwen and GLM, reaching 51.3% and 41.3% resolve rate while reducing latency to $0.88\times$ and $0.94\times$ full context, respectively. With Devstral, however, the same policy drops resolve rate to 38.7% and increases latency to $1.42\times$ full context. Thus, the impact of a given compression policy on both success and efficiency can change substantially across models.

Model-dependent policy rankings. The policy that performs best also changes across models. With Qwen, TRC achieves the highest resolve rate at 53.7%, the lowest billed cost at $0.71\times$, and the lowest latency at $0.79\times$ full context. With Devstral, TRC+SS achieves the highest resolve rate at 57.0%, while TR has the lowest latency at $0.90\times$ and SU the lowest billed cost at $0.85\times$. With GLM, OTRC and SU-p achieve the highest resolve rate at 41.3%, while SU has the lowest billed cost at $0.64\times$ but increases latency to $1.26\times$ and reduces resolve rate to 28.3%. The relative ordering of policies can even reverse across models: OTRC improves resolve rate over TR by 7.0 percentage points with Qwen, but reduces it by 17.0 points with Devstral. Therefore, no single compression policy performs best across models, and a policy that is effective for one model can be substantially worse for another.

Configuration sensitivity. Configuration effects also differ between Qwen and Devstral in the controlled sweeps, which vary threshold and depth while holding the primitive fixed. In our smaller ablation cohorts, increasing SS-p's SWE-bench trigger threshold from 10K to 15K to 20K increases Qwen's resolve rate from 50.7% to 60.0% to 73.3%, while Devstral remains roughly unchanged at 72–73% across its three thresholds. Compression depth shows a similar pattern: decreasing SU-p's depth from $D = 0.7$ to 0.5 to 0.3 on Terminal-Bench improves Qwen from 33.3% to 40.0% to 46.7%, but changes Devstral from 37.8% to 22.2% to 37.8%, respectively.

These comparisons establish model-dependent responses to compression, but do not isolate their causes. In particular, they do not distinguish differences in information retention, recovery from removed context, or execution behavior; thresholds are also calibrated separately for each model. Explaining the ranking reversals requires further controlled analysis.

Observation #6: While policy behavior is model-dependent, some design choices have consistent effects across models.

Although policy rankings differ across models, some design choices have consistent effects. Figure 5b compares these choices through paired policy comparisons; results below are reported in the order Qwen, Devstral, and GLM.

Shared design benefits. On SWE-bench, preserving recent history through partial structured rewriting (SS-p vs. SS) improves resolve rate by 6.7, 8.7, and 8.0 percentage points, respectively. Clearing tool outputs before structured rewriting (TRC+SS vs. SS) improves resolve rate by 7.7, 9.7, and 9.7 points. Both changes benefit all three models, providing consistent design guidance on this workload despite their different overall policy rankings.

Model-dependent effects. Other design choices do not show this consistency. Clearing tool outputs every step rather than only at the threshold (OTRC vs. TRC) changes SWE-bench resolve rate by -2.3, -14.3, and $+2.7$ percentage points. The same change therefore substantially hurts Devstral while having smaller, oppositely signed effects on Qwen and GLM.

Workload limits. Consistency across models does not guarantee transfer across workloads. Adding partial structured rewriting to per-step clearing (OTRC+SS-p vs. OTRC) reduces Terminal-Bench resolve rate by 12.5, 7.5, and 12.5 percentage points, while its effects on SWE-bench are mixed.

Takeaway 3. Compression policies are not one-size-fits-all across models, but their effects are not entirely model-specific either. Some design choices remain consistently beneficial across models, suggesting opportunities to combine reusable policy components with model- and workload-aware adaptation where needed.

5 Conclusion

We characterized the effects of context compression across two coding benchmarks and three open-weight models. Saving tokens did not reliably reduce latency or estimated input cost; the same compression policy could help on some tasks and hurt on others; and compression policy effects varied across models, although some design choices improved success for all three on SWE-bench. These findings motivate judging context compression by the executions it enables and tailoring policies to the task, model, and workload, with the goal of retaining the information an agent needs to finish efficiently.

AI USE STATEMENT

In this work, we used generative AI tools for to assist with implementing context compression policies, developing experiment and analysis scripts and generating plots. We have not used generative AI tools for to develop the conceptual framework, formulate claims, design the methodology or experiments, and the remaining categories in the ICLR policy are not applicable.

REPRODUCIBILITY STATEMENT

We document compression implementations and summarization prompts in Appendix A.2, task cohorts and experimental configurations in Appendix A.3, and threshold calibration in Appendix A.4. Appendix A.5 specifies metric definitions, run-inclusion rules, task pairing, and statistical procedures. Appendix A.6.4 reports run outcomes and resource-limit handling to clarify how termination behavior affects the interpretation of our results.

REFERENCES

Anthropic. Context editing. https://platform.claude.com/docs/en/build-with-claude/context-editing, 2025. Accessed 2026-05-11.

Anthropic. Claude code documentation, context window and model configuration. https://code.claude.com/docs/en/context-window, https://code.claude.com/docs/en/model-config, 2026a. Accessed 2026-09-10.

Anthropic. Compaction. https://platform.claude.com/docs/en/build-with-claude/compaction, 2026b. Accessed 2026-05-11.

Anthropic. Prompt caching. https://platform.claude.com/docs/en/build-with-claude/prompt-caching, 2026c. Accessed 2026-09-25. Cache reads billed at 0.1x the base input price (0.025x on Claude Fable 5.1 / Mythos 5.1).

Chaokun Chang, Yukun Zhou, Kaihua Fu, Dakai An, Tianyu Feng, Hanfeng Lu, Sheng Yao, Pu Guo, Yinghao Yu, Yizhou Shan, Bo Li, Binhang Yuan, and Wei Wang. From llm inference to agentic workloads: Characterization and implications for serving systems, 2026. URL https://arxiv.org/abs/2608.15127.

Haorui Chen, Yuancheng Zhu, Yitong Zhang, and Jia Li. Coact: Action-preserving observation compression for coding agents. *arXiv preprint arXiv:2607.02911*, 2026.

Cline. Auto compact. https://docs.cline.bot/features/auto-compact, 2026. Accessed 2026-09-03. Strategies `agentic`, `basic`, `off` via `--compaction`; trigger at 90% of the window.

Yufeng Du, Minyang Tian, Srikanth Ronanki, Subendhu Rongali, Sravan Bodapati, Aram Galstyan, Azton Wells, Roy Schwartz, Eliu A Huerta, and Hao Peng. Context length alone hurts llm performance despite perfect retrieval. *arXiv preprint arXiv:2510.05381*, 2025.

In Gim, Guojun Chen, Seung seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. Prompt cache: Modular attention reuse for low-latency inference, 2024. URL https://arxiv.org/abs/2311.04934.

Google. Gemini CLI configuration reference. https://geminicli.com/docs/reference/configuration/, 2026a. Accessed 2026-09-03. `model.compressionThreshold` default 0.5. The 30% retention is the constant `COMPRESSION_PRESERVE_THRESHOLD = 0.3` in https://github.com/google-gemini/gemini-cli/blob/main/packages/core/src/context/chatCompressionService.ts.

Google. Context caching. https://ai.google.dev/gemini-api/docs/caching, 2026b. Accessed 2026-09-25. Cached input tokens billed at a reduced rate.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32779–32798, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*, 2025.

Donghwan Kim, Prakhar Singh, Younghoon Min, Jongryool Kim, Jongse Park, and Kiwan Maeng. Characterization of multi-model agentic ai systems on general tasks via trace-driven simulation, 2026a. URL https://arxiv.org/abs/2606.01725.

Jiin Kim, Byeongjun Shin, Jinha Chung, and Minsoo Rhu. The cost of dynamic reasoning: Demystifying ai agents and test-time scaling from an ai infrastructure perspective, 2026b. URL https://arxiv.org/abs/2506.04301.

Leonid Kondrashov, Hongrui Liu, JooYoung Park, Boxi Zhou, Zonghao Liu, Chengzhi Lu, Riccardo Mancini, Esha Choukse, Haris Javaid, German Sviridov, Tao Peng, Chen Zhao, Anastasia Avdeeva, Aleksei Gusev, Marios Kogias, Luo Mai, and Dmitrii Ustiugov. Rethinking ai cloud infrastructure for agentic serving systems with the aries experimentation framework, 2026. URL https://arxiv.org/abs/2607.29069.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. Llms get lost in multi-turn conversation. In *International Conference on Learning Representations*, volume 2026, pp. 54738–54778, 2026.

LangChain. Short-term memory — trim messages. https://docs.langchain.com/oss/python/langchain/short-term-memory, 2025. Accessed 2026-05-11.

Mosh Levy, Alon Jacoby, and Yoav Goldberg. Same task, more tokens: the impact of input length on the reasoning performance of large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15339–15353, 2024.

Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. The complexity trap: Simple observation masking is as efficient as llm summarization for agent context management. *arXiv preprint arXiv:2508.21433*, 2025.

Banruo Liu, Haoran Qiu, Íñigo Goiri, Rodrigo Fonseca, Ricardo Bianchini, and Esha Choukse. Agentic coding in the wild: Characterizing github copilot traces at production scale. *arXiv preprint arXiv:2608.00101*, 2026.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173, 2024.

Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, and Junchen Jiang. Lmcache: An efficient kv cache layer for enterprise-scale llm inference, 2025. URL https://arxiv.org/abs/2510.09665.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

Microsoft. Manage agent sessions in VS Code: Context compaction. https://code.visualstudio.com/docs/agents/run/sessions/manage-sessions, 2026. Accessed 2026-09-03. Compaction fires when the context window is filled and summarizes all preceding turns in a separate model call. Copilot CLI variant (80% threshold, checkpoint file): https://docs.github.com/en/copilot/concepts/agents/copilot-cli/context-management.

Guanghui Min, Liang Wu, Mayank Darbari, Chen Chen, and Liangjie Hong. Toward reliable context compression for long-horizon agents: An empirical study of execution instability. *arXiv preprint arXiv:2608.06503*, 2026.

OpenAI. Responses api — create response. https://platform.openai.com/docs/api-reference/responses/create, 2025. Accessed 2026-05-11.

OpenAI. Codex configuration reference. https://learn.chatgpt.com/docs/config-file/config-reference, 2026a. Accessed 2026-09-03 (redirect target of https://developers.openai.com/codex/). model_auto_compact_token_limit, derived as 90% of the context window when unset. Server-side compaction: https://developers.openai.com/api/docs/guides/compaction.

OpenAI. Compaction. https://developers.openai.com/api/docs/guides/compaction, 2026b. Accessed 2026-05-11.

OpenAI. Prompt caching. https://developers.openai.com/api/docs/guides/prompt-caching, 2026c. Accessed 2026-09-25. Cached input tokens discounted up to 90% relative to uncached input.

OpenHands. Context condenser: OpenHands software agent SDK architecture. https://docs.openhands.dev/sdk/arch/condenser, 2026. Accessed 2026-09-03. Default LLMSummarizingCondenser; "replacing the first half of all events with a single summary event". Source: https://github.com/OpenHands/software-agent-sdk/blob/main/openhands-sdk/openhands/sdk/context/condenser/llm_summarizing_condenser.py.

Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. Memgpt: towards llms as operating systems. 2023.

Pi Coding Agent. Context compaction. https://pi.dev/docs/latest/compaction, 2026. Accessed 2026-09-03. Auto-compaction when contextTokens > contextWindow - reserveTokens Algorithm: https://github.com/earendil-works/pi/blob/main/packages/coding-agent/docs/compaction.md.

Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving, 2025. URL https://arxiv.org/abs/2407.00079.

Ritik Raj, Souvik Kundu, Ishita Vohra, Hong Wang, and Tushar Krishna. Towards understanding, analyzing, and optimizing agentic ai execution: A cpu-centric perspective, 2026. URL https://arxiv.org/abs/2511.00739.

Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for llm serving, 2024. URL https://arxiv.org/abs/2407.00023.

Ningzhi Tang, Chaoran Chen, Gelei Xu, Yiyu Shi, Yu Huang, Collin McMillan, Tao Dong, and Toby Jia-Jun Li. How coding agents fail their users: A large-scale analysis of developer-agent misalignment in 20,574 real-world sessions. *arXiv preprint arXiv:2605.29442*, 2026.

Runchu Tian, Yanghao Li, Yuepeng Fu, Siyang Deng, Qinyu Luo, Cheng Qian, Shuo Wang, Xin Cong, Zhong Zhang, Yesai Wu, et al. Distance between relevant information pieces causes bias in long-context llms. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 521–533, 2025.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. Reducing cost of llm agents with trajectory reduction. *Proceedings of the ACM on Software Engineering*, 3(FSE):1241–1263, 2026.

Buqiang Xu, Zirui Xue, Dianmou Chen, Chenyang Fu, Chiyu Wu, Caiying Huang, Chen Jiang, Jizhan Fang, Xinle Deng, Yijun Chen, et al. Tokenpilot: Cache-efficient context management for llm agents. *arXiv preprint arXiv:2606.17016*, 2026.

Jirong Yang, Peizhe Liu, Chaojie Zhang, and Jovan Stojkovic. Architectural implications of agentic ai workflows, 2026. URL https://arxiv.org/abs/2608.04458.

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. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Yichao Yuan, Mosharaf Chowdhury, and Nishil Talati. Kairos: Stateful, context-aware power-efficient agentic inference serving, 2026a. URL https://arxiv.org/abs/2604.16682.

Yichao Yuan, Ankita Nayak, Souvik Kundu, and Nishil Talati. Agentic ai workload characteristics, 2026b. URL https://arxiv.org/abs/2605.26297.

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.

Yusheng Zheng, Jiakun Fan, Quanzhi Fu, Yiwei Yang, Wei Zhang, and Andi Quinn. Agentcgroup: Understanding and controlling os resources of ai agents, 2026. URL https://arxiv.org/abs/2602.09345.

A APPENDIX

Appendix A.1 reviews related work. Appendix A.2 details the compression implementations, summarizer prompts, and mappings to deployed systems. Appendices A.3 and A.4 describe the experimental configurations and threshold calibration. Appendix A.5 defines the metrics and statistical procedures. Appendix A.6 presents threshold and depth sweeps, task-level gains and losses, summarizer sensitivity, run outcomes and resource limits, and changes in execution length.

A.1 RELATED WORKS

Prior agentic serving characterization. Long agent trajectories can become difficult to serve effectively even before reaching the model's context limit. Prior work shows that models make weaker use of distant information and often incur increasing token overhead as context grows (Packer et al., 2023; Hu et al., 2025; Liu et al., 2024; Tian et al., 2025; Du et al., 2025). Recent characterization studies examine related effects at different levels: multi-turn performance degradation in LLMs (Laban et al., 2026), end-to-end execution (Kim et al., 2026b; Raj et al., 2026; Chang et al., 2026), resource usage (Yang et al., 2026; Zheng et al., 2026), and serving behavior of agents (Kondrashov et al., 2026; Kim et al., 2026a; Yuan et al., 2026a;b), and production-scale workload, context growth, and cache behavior in GitHub Copilot (Liu et al., 2026). Together, these works establish that growing agent context affects both model behavior and serving efficiency. However, they characterize workloads under a given context-management mechanism rather than systematically studying the compression policy itself. Our work instead varies *what* is compressed, *when* compression is triggered, and *how aggressively* history is compressed, and measures how these choices change task success, execution length, cost, and latency.

Context management inside the agent loop. Modern agents increasingly manage growing histories by truncating observations, summarizing earlier interactions, or combining both (OpenAI, 2026a; Anthropic, 2025; Google, 2026a; Microsoft, 2026; OpenHands, 2026; Cline, 2026). These mechanisms are already consequential in production: across 13 million Copilot sessions, sessions that invoke compaction account for 44% of served tokens, and the median compaction removes 73% of the context (Liu et al., 2026).

Recent work proposes compression techniques that substantially reduce retained context while often preserving task success (Kang et al., 2025; Chen et al., 2026; Lindenbauer et al., 2025; Xiao et al., 2026). However, compression can also alter the execution itself: agents may take additional steps, revisit discarded information, or incur extra model calls to construct summaries (Min et al., 2026; Lindenbauer et al., 2025). Observational studies of deployed coding agents also identify failures to carry forward earlier constraints and decisions (Tang et al., 2026). Although these failures are not attributed specifically to compression, they motivate examining what information context-management policies preserve. Consequently, evaluating compression only by the number of tokens removed can miss important downstream effects. We build on this observation by treating compression as a policy with separable design choices (primitive, trigger, and depth) and studying how each shapes the resulting agent trajectory across models and task environments.

Serving efficiency and end-to-end cost. Prompt size is also an incomplete proxy for serving cost. Modern serving systems reuse KV state (Kwon et al., 2023; Qin et al., 2025) and shared prompt prefixes (Zheng et al., 2024; Gim et al., 2024; Srivatsa et al., 2024) to reduce the cost of repeated context, while commercial APIs similarly discount cached input tokens (Anthropic, 2026c; OpenAI, 2026c; Google, 2026b). Compression can disrupt this reuse: rewriting earlier history may invalidate a cached prefix (Liu et al., 2025), while summarization itself introduces additional inference. Production Copilot traces observe reduced cache reuse around compaction (Liu et al., 2026), and TokenPilot proposes prefix-stable compaction specifically to preserve reuse (Xu et al., 2026). As a result, reducing prompt tokens does not necessarily reduce either billed cost or end-to-end latency.

Our study connects these effects in a controlled evaluation of the compression policy itself. Rather than treating compression efficiency as the size of the compressed prompt, we evaluate the full execution it induces, including task outcome, token usage, number of agent steps, serving cost, and latency.

A.2 PRIMITIVE IMPLEMENTATION

Shared rules. The system prompt and initial task instruction are retained verbatim; compression operates on the remaining history. Before each agent call, threshold policies count message-content tokens using `tiktoken` (`cl100k_base`) and fire when the estimate exceeds B. This estimate excludes model-specific chat-template overhead.

Truncation (TR). TR removes the oldest whole messages until the target is reached, always retaining the final compressible message. It does not preserve action–response pairs, and an oversized final message can leave the context above target.

Tool-result clearing (TRC). TRC replaces older tool responses with short placeholders, retaining surrounding assistant messages and the three most recent eligible responses. It skips existing clearing and summary markers. Clearing proceeds oldest first until the target is reached; if insufficient, standalone TRC falls back to TR. This fallback can remove recent responses that were protected from clearing.

Summarization (SU and SS). Both replace the compressible history with one user-role summary. SU requests free-form prose covering the task objective, progress, files examined or modified, observations, errors, decisions, and remaining work. SS instead requires five ordered sections: Task, Files Modified, Files Examined, Execution Anchors, and Current State. Both prompts emphasize preserving exact paths and relevant errors; SS additionally requests exact commands, test results, and symbol names and prohibits unsupported additions. The summarizer receives a role-tagged transcript of the compressible history, including any previous summary, but not the protected prefix.

Partial summarization (SU-p and SS-p). Partial variants allocate half the available token allowance to a verbatim recent tail and half to a summary of the older head. The tail is the longest contiguous suffix of whole messages that fits its allowance; it may be empty. The head uses the corresponding SU or SS prompt. If the reconstructed context exceeds the target, TR removes the oldest messages, potentially including the summary.

Step-triggered clearing (OTRC). Before each call, OTRC clears the response preceding the four most recent tool responses in an alternating action–response history. The implementation replaces the ninth message from the end once at least five calls and ten messages are present. This positional rule also applies after stacked rewrites, without checking message roles or summary markers. Standalone OTRC has no token threshold.

Summary prompts request approximately 0.75 words per allocated token, with a minimum allowance of 50 tokens. By default, the summarizer uses the agent model, temperature 0.2, and a 4096-token output cap. Summary length is a soft target: SU has no length fallback, and SS's TR fallback cannot shorten its sole retained summary message. The exact SU and SS prompt templates are reproduced below.

A.2.1 SUMMARIZER PROMPT TEMPLATES

The system and user messages below reproduce the templates in `memory.py`, with line wrapping for readability. At runtime, {`target_words`} is replaced by the requested word count and {`history_text`} by the role-tagged transcript. SU-p and SS-p use the same templates with only the older head as the transcript.

SU: System message.

```
You are summarizing an agent's work history to free up context
window space. Your output will replace the conversation history
| the agent will only see your summary, so it must be complete
enough to continue the task.
```

SU: User message.

```
Summarize the following agent conversation history in
```

```
approximately
{target_words} words. Your summary MUST include all of:
1. Current task objective and progress made so far
2. Files examined and any modifications made (include exact file
paths)
3. Key observations, errors encountered, and decisions taken
4. Current state | what has been done and what remains

Preserve exact file paths, error messages, and code snippets
that are likely still relevant. Be factual and concise.

{history_text}
```

SS: System message.

```
You are compressing an agent's working memory to free up context
window space. Your output will REPLACE the entire conversation
history | the agent will only see your summary. It must contain
everything needed to continue the task without any other
context.
```

SS: User message.

```
Produce a structured summary of the agent conversation below in
approximately
{target_words} words total. Use EXACTLY this format and section
order:

[CONTEXT SUMMARY]
## Task
<One sentence: what the task is asking for.>

## Files Modified
<Each file the agent has written or edited. Format: path | what
changed and why. If none yet, write 'None'.>

## Files Examined
<Each file the agent has read. Format: path | key observation.
If none yet, write 'None'.>

## Execution Anchors
<Exact commands run, test results, error messages, or shell
output that the agent will need to reference going forward.
Quote verbatim where possible.>

## Current State
<What has been done, what has NOT been done, and the immediate
next step.>
[END CONTEXT SUMMARY]

Rules:
- Preserve exact file paths, line numbers, error strings, and
symbol names.
- Do not speculate or add information not present in the
history.
- Do not add any text outside the [CONTEXT SUMMARY] block.

Conversation history:
{history_text}
```

A.2.2 Compression in deployed systems

Table 2 maps representative deployed compression mechanisms onto our primitive–trigger–depth design space. These mappings identify the closest policy analogues; implementation details may differ from our evaluated policies.

System	What is removed (P)	When it fires (T)	How much (D)	User sets	$P/T/D$
OpenHands (2026)	Older half of the log $\rightarrow$ free-form summary	Th at 80 events (default) or a token cap	$D = 0.5$; keeps first few and latest events	P, T, D	$SU\text{-}p/Th/0.5$
Gemini CLI (2026a)	Tool results truncated; older 70% $\rightarrow$ structured summary	Th at 0.5 of the context window	$D = 0.7$ (fixed); last 30% survives	T	$TRC{+}SS\text{-}p/Th/0.7$
Claude Code (2026a)	Tool outputs cleared; history $\rightarrow$ structured summary	Th at the context limit (0.97 on a 1M window)	$D \rightarrow 1$; nothing kept verbatim	T	$TRC{+}SS/Th/1.0$

Table 2: Three deployed compression mechanisms placed in the three-axis design space. In the P column $\rightarrow$ reads as replaced by, and a semicolon separates the stages of a stacked primitive. Th denotes a threshold trigger, given as an event count or a window fraction. D is the fraction of context removed, after both stages. The last column gives the closest analogue in our notation rather than an exact match. All three fire on a threshold, and their depths run from 0.5 to 1, so our canonical depth sits on a deployed default. See §2.

A.3 Experimental Configurations

Tasks and evaluation subsets. SWE-bench Verified evaluates repository-level bug fixing on real GitHub issues. Our 100 issues come from Django (34), SymPy (34), and scikit-learn (32), which cover web development, symbolic mathematics, and machine learning. For Terminal-Bench 1.0 we use the `terminal-bench-core` 0.1.1 dataset. Its 40 tasks cover build and environment setup, data processing, machine-learning workflows, security, system administration, and version control. The main comparisons use all selected tasks. The threshold and depth sweeps use subsets of 25 SWE-bench and 15 Terminal-Bench tasks to limit the cost of evaluating many configurations (Table 3).

Benchmark	Main comparison	Threshold/depth sweeps
SWE-bench Verified	100	25
Terminal-Bench	40	15

Table 3: Number of tasks in each evaluation. Sweep tasks are subsets of the main comparison tasks. The SWE-bench sweep subset contains 9 Django, 8 SymPy, and 8 scikit-learn issues.

Harness and serving. We use a modified mini-swe-agent harness with three open-weight models: Qwen3.5-35B-A3B, Devstral-Small-2-24B, and GLM-4.7-Flash. The models span a dense transformer (Devstral), a mixture-of-experts transformer (GLM), and a mixture-of-experts model combining linear and full attention (Qwen), allowing us to examine whether compression effects transfer across models.

All models are served locally through vLLM (Kwon et al., 2023), on four A100 80GB GPUs for SWE-bench and four RTX A6000 48GB GPUs for Terminal-Bench, using temperature 0.2 and a 4096-token output cap (Table 5).

Experiment grid. Table 4 summarizes the full comparison for all three models and the controlled sweeps for Qwen and Devstral on both benchmarks. GLM is evaluated only at the primary threshold and fixed depth $D = 0.5$, alongside the FC and OTRC references. Model-specific trigger thresholds are given in Appendix A.4.

The summarizer comparison uses the 25-task SWE-bench subset and Qwen's 15K threshold to compare self-summarization with Qwen3.5-9B and Gemma-4-12B for SU and TRC+SU (Appendix A.6.3).

A.4 Trigger-Threshold Calibration

We calibrate thresholds separately for each benchmark and model using uncompressed trajectories. For each trajectory, we record its peak context: the largest server-reported prompt-token count across its calls. We select tight, primary, and loose thresholds near the 5th, 15th, and 25th percentiles of the peak-context distribution, rounding to the nearest 1K tokens. This places compression relative to the context demands of each workload: lower thresholds trigger earlier and affect more tasks. The

	Depth-tunable					Depth-invariant						Reference	
Trigger threshold	TR	SU	SU-p	SS	SS-p	TRC	TRC+SU	TRC+SS	OTRC+TR	OTRC+SU-p	OTRC+SS-p	FC	OTRC
Trigger T	Th	Th	Th	Th	Th	Th	Th	Th	$St{+}Th$	$St{+}Th$	$St{+}Th$	–	St
Tight	○ ○ ○	○ ○ ○	○ ○ ○	○ ○ ○	○ ○ ○	○	○	○	○	○	○	–	–
Primary	○ ● ○	○ ● ○	○ ● ○	○ ● ○	○ ● ○	●	●	●	●	●	●	–	–
Loose	○ ○ ○	○ ○ ○	○ ○ ○	○ ○ ○	○ ○ ○	○	○	○	○	○	○	–	–
No threshold	–	–	–	–	–	–	–	–	–	–	–	●	●

Legend: ● Full comparison; ○ Controlled sweeps; – not applicable.
SWE-bench: 100 / 25 tasks; Terminal-Bench: 40 / 15 tasks (full / sweeps).
●: all three models; ○: Qwen and Devstral only.

Table 4: Experiment grid for both benchmarks. GLM includes only the filled marks; Qwen and Devstral include both filled and open marks. For depth-tunable policies, the three marks read left to right as $D = 0.3, 0.5, 0.7$; the full comparison uses the primary threshold and $D = 0.5$. Depth-invariant policies use one fixed configuration per threshold, as described in Appendix A.2. FC and standalone OTRC are evaluated once per task repetition without an operative token threshold. Each marked configuration targets three runs per task. Full-comparison runs on the sweep subsets also provide the corresponding primary-threshold comparisons.

Table 5: Software environment and inference settings. Unless noted, values are shared across Qwen3.5-35B-A3B, Devstral-24B, and GLM-4.7-Flash.

Setting	**Value**
Software	
Python	3.10.12
CUDA	12.3 (Qwen, Devstral) / 12.9 (GLM)
PyTorch	2.10.0 (Qwen, Devstral) / 2.13.0 (GLM)
Transformers	4.57.6 (Qwen, Devstral) / 5.16.1 (GLM)
vLLM	0.17.1 (Qwen, Devstral) / 0.28.0 (GLM)
mini-swe-agent	2.2.6
swe-bench	4.1.0
terminal-bench-core	0.1.1 (Terminal-Bench 1.0)
Harbor	0.20.0
Inference	
GPUs	4× A100 80GB (SWE-bench) / 4× RTX A6000 48GB (Terminal-Bench)
dtype	bf16
Tensor parallel	4
max-model-len	102,400 (Qwen) / 65,536 (Devstral, GLM)
max-num-seqs	64
Concurrency	16 (SWE-bench) / 4 (Terminal-Bench)
Temperature	0.2

full comparison uses the primary threshold; controlled sweeps use all three thresholds for Qwen and Devstral. GLM uses only the primary threshold.

Table 6 lists the thresholds. Qwen's SWE-bench settings predate the percentile rule and correspond approximately to the 5th, 16th, and 28th percentiles. Calibration uses server-reported counts, while runtime checks use the token estimate described in Appendix A.2; exact crossing rates can therefore differ.

A.5 Metrics and statistical comparisons

Each task–policy–model configuration is repeated three times. Task success is determined by the benchmark's native evaluator. Token consumption includes prompt and output tokens from both agent and compression calls; latency measures wall-clock time from trajectory start to termination. Billed input cost is estimated as described in Appendix A.5.2. We examine aggregate performance

Benchmark	Model	Tight	Primary	Loose
SWE-bench Verified	Qwen3.5-35B-A3B	10K	15K	20K
	Devstral-Small-2-24B	17K	21K	24K
	GLM-4.7-Flash	10K	13K	15K
Terminal-Bench	Qwen3.5-35B-A3B	2K	3K	4K
	Devstral-Small-2-24B	3K	4K	7K
	GLM-4.7-Flash	–	3K	–

Table 6: Trigger thresholds by benchmark and model, in tokens (K = 1,000). On Terminal-Bench, GLM is evaluated only at the primary threshold.

and individual-task outcomes, and compare policy effects relative to each model's full-context baseline when assessing transfer across models.

Success and task-level comparisons. Resolve rate averages each task's success fraction across three runs. Runs terminated by resource limits without a successful outcome count as unresolved. For task gains and losses, we consider a task solved when at least two of its three runs succeed. A gain is a task solved by the policy but not FC; a loss is the reverse. These comparisons use the full cohorts (100 SWE-bench tasks and 40 Terminal-Bench tasks), require three recorded attempts per task and policy, and treat missing verdicts as unresolved. Step comparisons use only successful attempts on tasks solved by both policies, averaging steps within each task before comparing task means.

Resource comparisons. Main comparisons of token consumption, cost, and latency exclude runs that reach the time limit (1500 seconds on SWE-bench or the task-specific Harbor timeout on Terminal-Bench). Runs that reach only the step limit are included. Resource accounting requires at least two recorded prompt-token counts per run. We average eligible runs within each task and retain tasks with measurements for both the policy and FC. The reported ratio divides the policy's average across these tasks by FC's average. The controlled-sweep table in Figure 3 (right) includes all attempts in its cost and latency averages; the curves on the left exclude time-capped runs.

Uncertainty. We report 95% confidence intervals by repeatedly sampling tasks with replacement. Each sampled task retains all its runs for both the policy and FC. We recalculate the success difference or resource and step ratios for each sample.

A.5.1 Token consumption

Token consumption is the total number of prompt and output tokens recorded over a run, including the summarization calls to the model for compression. Peak context measures the largest prompt in a single call; cumulative consumption sums usage across calls. A trigger threshold therefore does not bound total token consumption.

Figure 6 illustrates this distinction for two runs of the same task: shorter prompts and fewer calls both reduce cumulative prompt consumption.

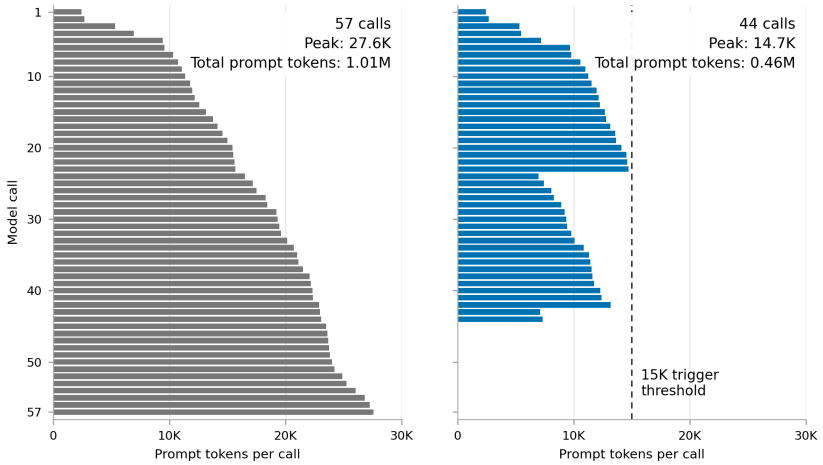
A.5.2 Billed cost calculation

We estimate billed input cost in units of one fresh input token:

$$\widehat{C}_{\text{input}} = N_{\text{fresh}} + 0.1\, N_{\text{cached}} + N_{\text{summary}} \tag{1}$$

where N_{fresh} and N_{cached} are the reconstructed fresh and cached agent input-token counts, and N_{summary} is the total summarizer input-token count. Cached input is charged at one-tenth the fresh-input rate; summarizer input is treated as fresh. Output tokens contribute to token consumption but are excluded from this input-cost metric.

We estimate fresh and cached input-token counts from successive prompt sizes and recorded OTRC events, using 16-token blocks. For prompt reductions of at least $\max(1000, 0.25P)$ tokens, where P is the preceding prompt size, tokens beyond the estimated stable prefix are counted as fresh.



(a) Full context (b) Truncation

Figure 6: Prompt tokens per model call for Qwen3.5-35B-A3B on scikit-learn 13142, run 1: full context (left) and truncation with a 15K trigger threshold (right). Each bar represents one call; cumulative prompt consumption is the sum of the bar lengths.

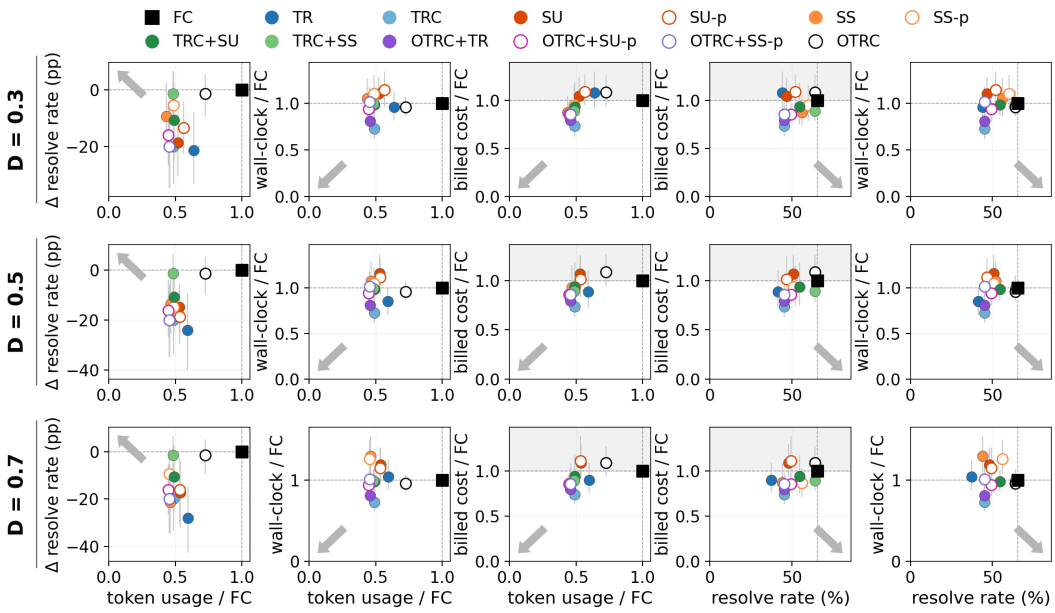
For OTRC events, tokens from the cleared message onward are counted as fresh; ordinary prompt growth is also charged as fresh input. For smaller unexplained reductions, we use the conservative reconstruction that charges the suffix beyond the stable prefix as fresh. This is an estimate under the assumed cache discount, not a measured provider bill or cache-hit rate; Qwen was served with prefix caching disabled.

A.6 Extended Results

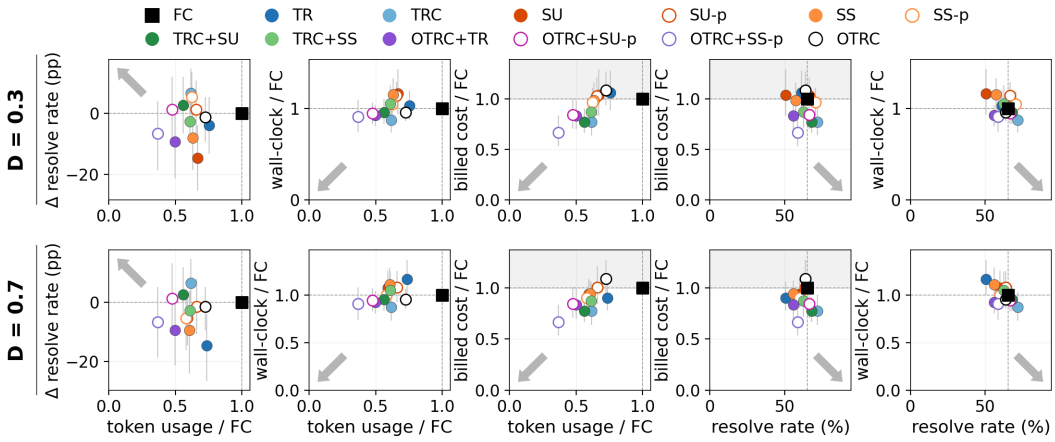
We report trigger-threshold and depth sweeps, task-level gains and losses, and sensitivity to the summarizer model.

A.6.1 Trigger-threshold and depth sweeps

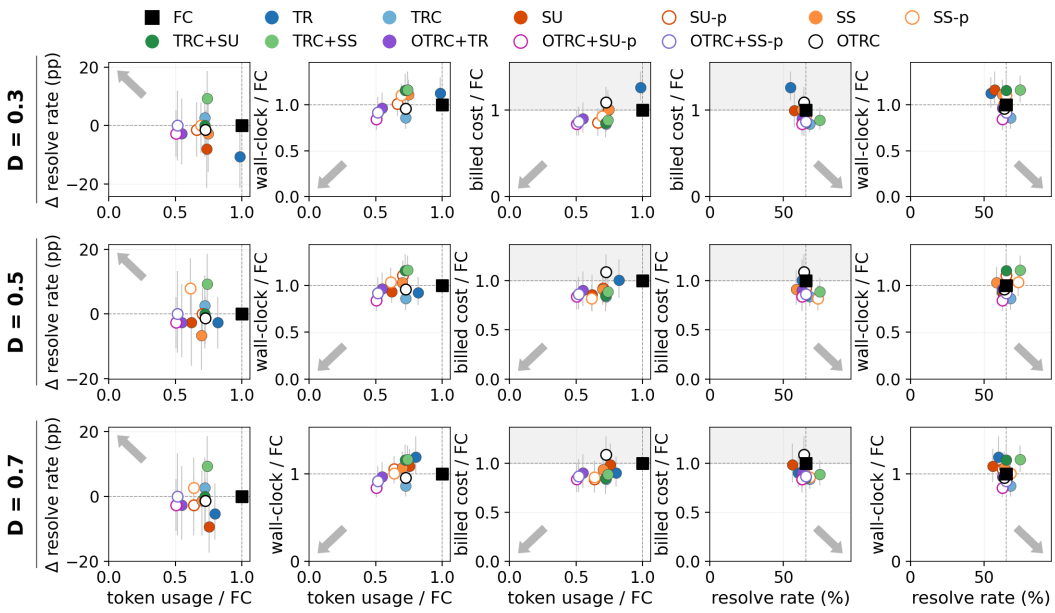
Figures 7–9 extend the main comparison to the controlled-sweep subsets. Each subfigure fixes one trigger threshold and compares success, token consumption, latency, and billed input cost across compression depths.



(a) Tight threshold (10K)

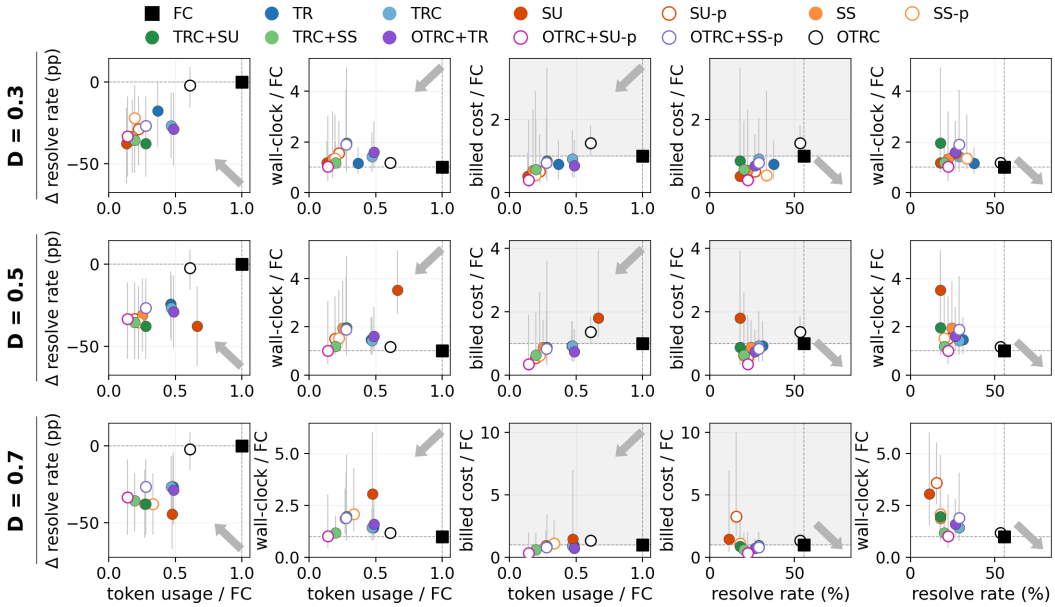


(b) Primary threshold (15K)

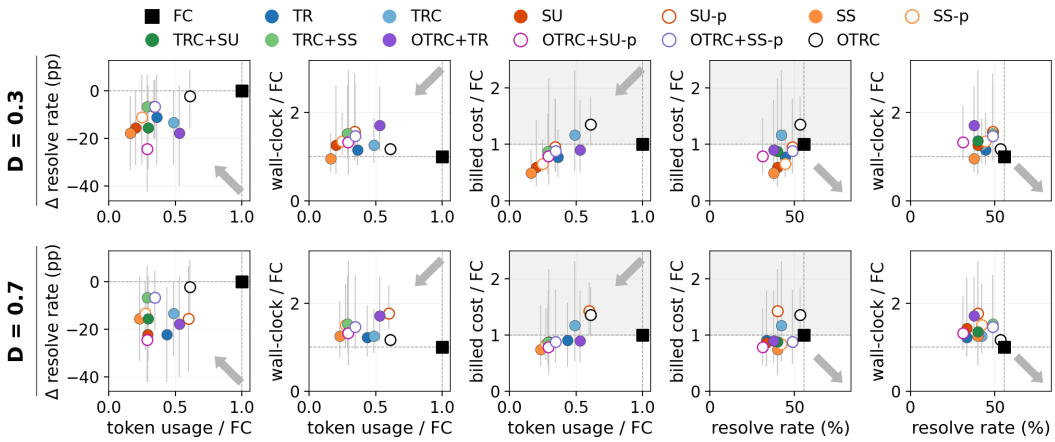


(c) Loose threshold (20K)

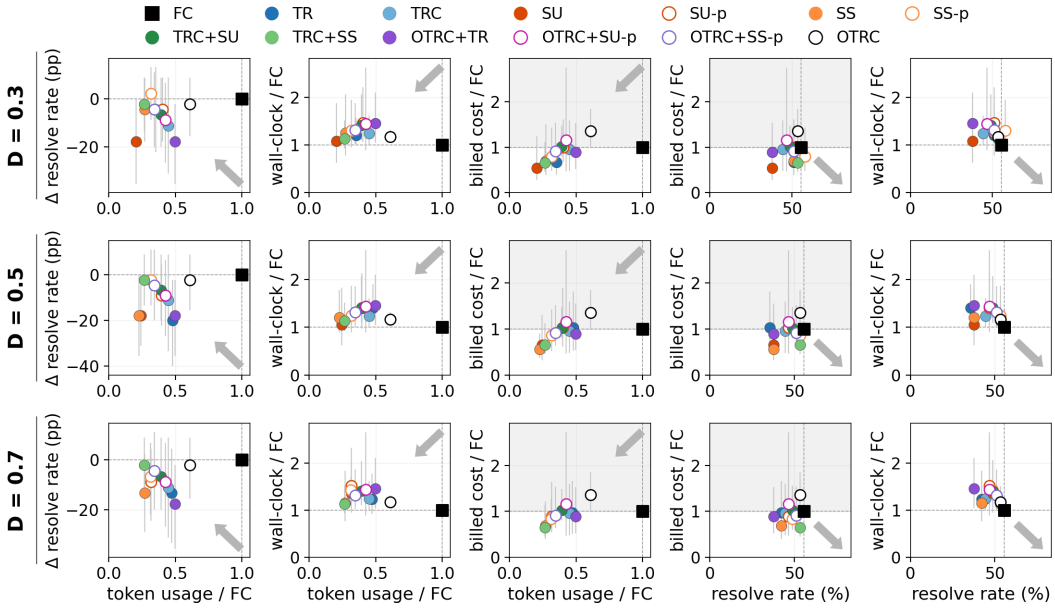
Figure 7: Depth and threshold ablation for Qwen3.5-35B-A3B on SWE-bench.



(a) Tight threshold (2K)

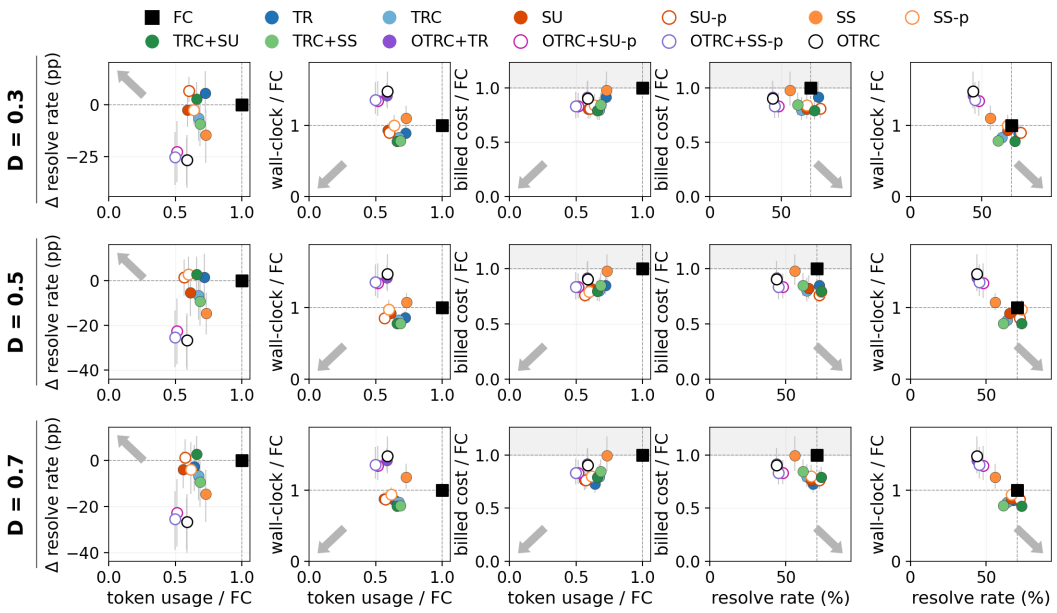


(b) Primary threshold (3K)

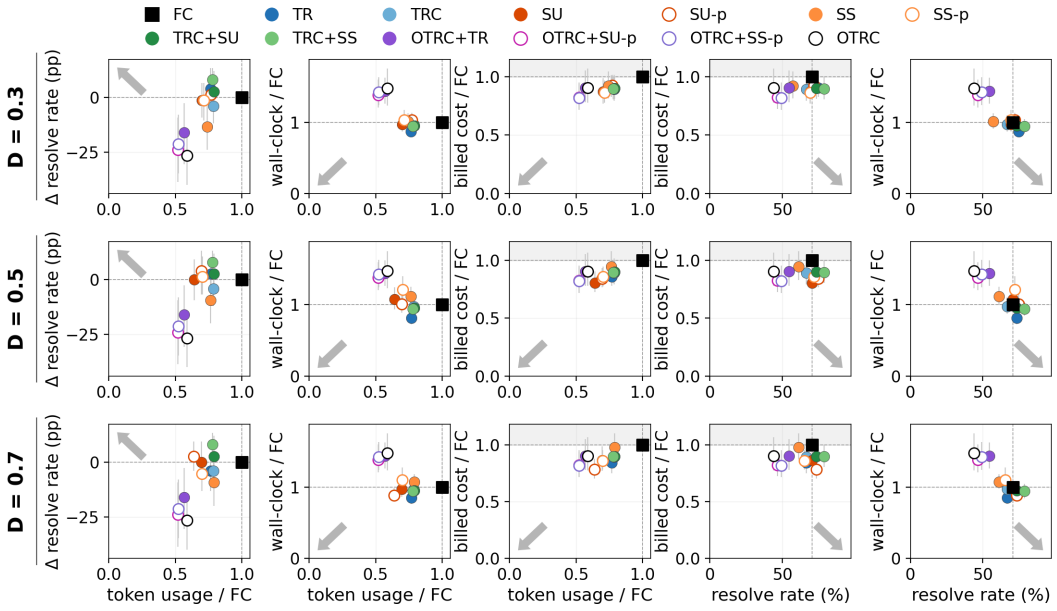


(c) Loose threshold (4K)

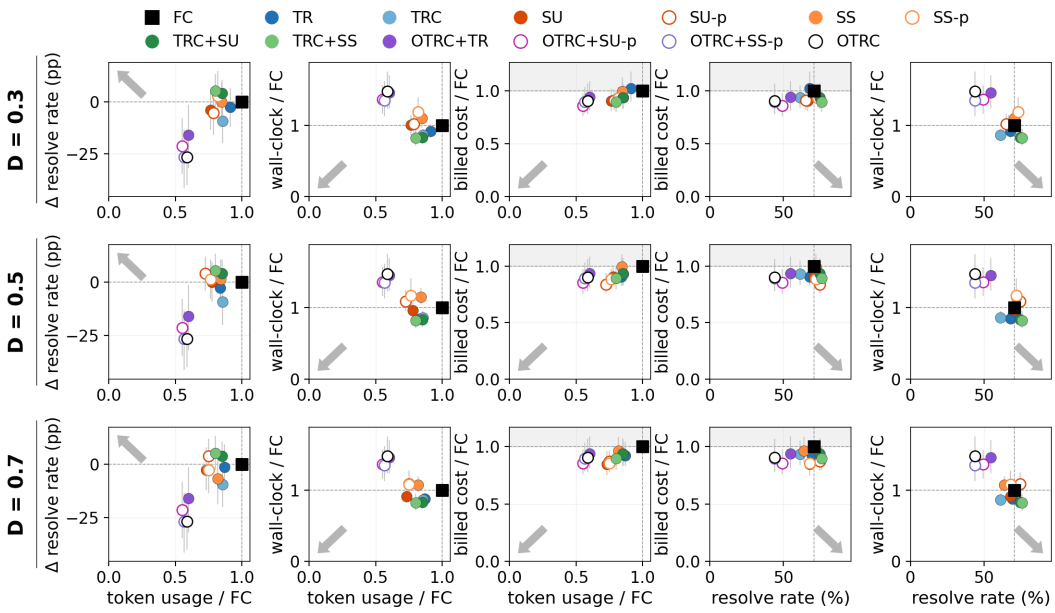
Figure 8: Depth and threshold ablation for Qwen3.5-35B-A3B on Terminal-Bench.



(a) Tight threshold (17K)



(b) Primary threshold (21K)



(c) Loose threshold (24K)

Figure 9: Depth and threshold ablation for Devstral-Small-2-24B on SWE-bench.

A.6.2 Task-level gains and losses

Figures 10–12 extend Figure 4 to the 25-task SWE-bench subset across trigger thresholds and compression depths. Each panel shows which tasks Qwen gains or loses relative to FC, using the success criterion in Appendix A.5.

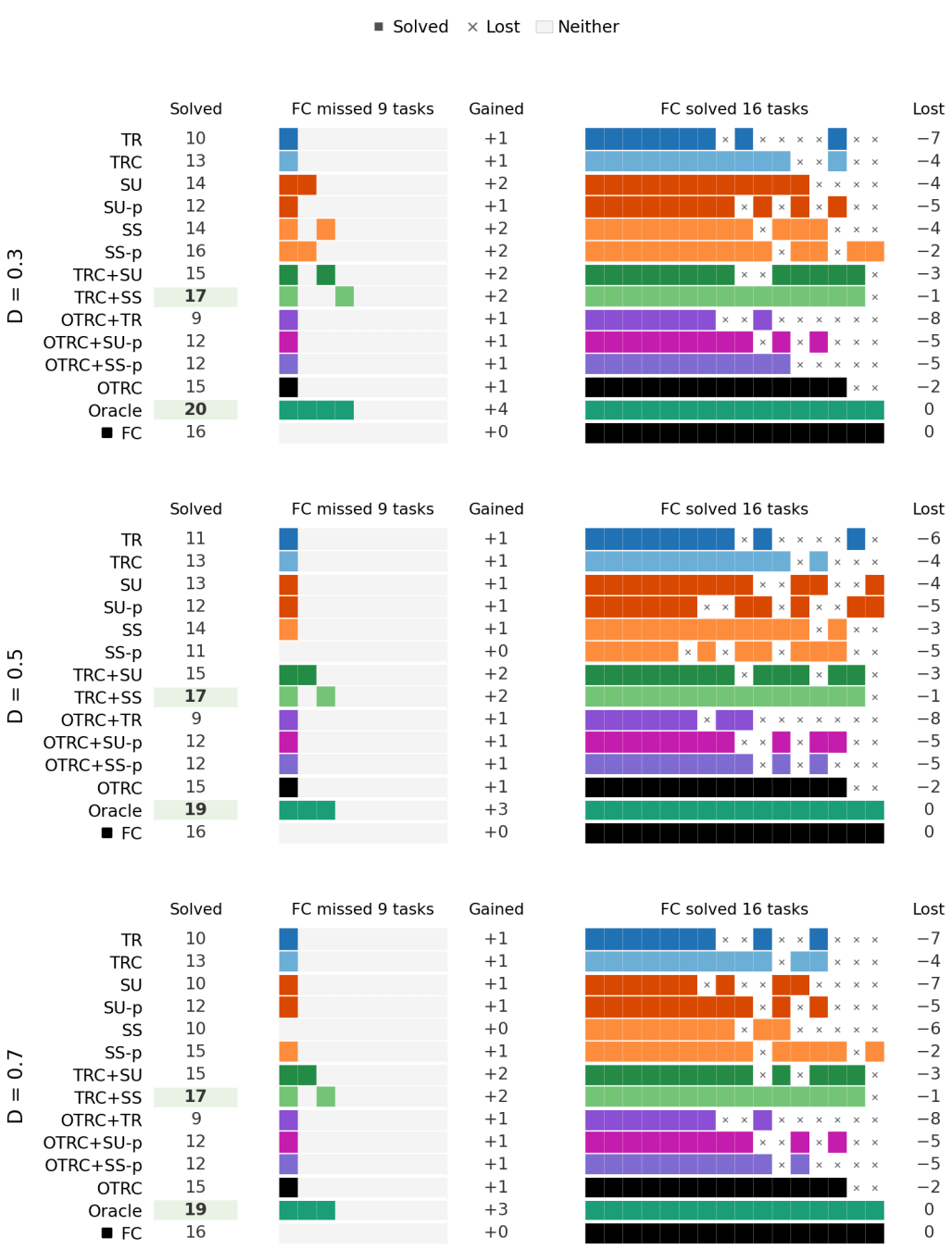


Figure 10: **Tight threshold (10K).** Task coverage relative to full context for Qwen3.5-35B-A3B on SWE-bench (threshold/depth sweeps, 25 tasks).

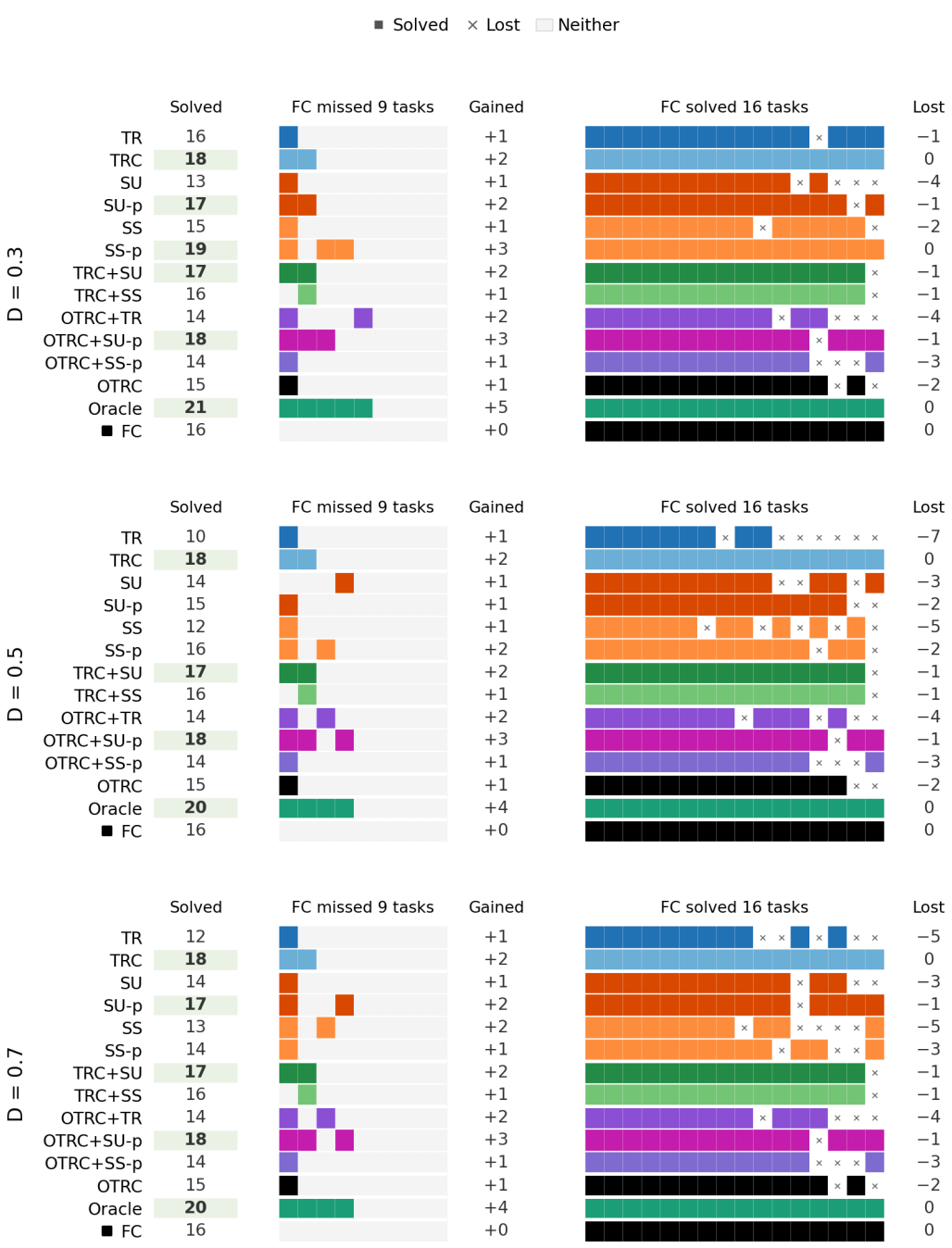


Figure 11: **Primary threshold (15K).** Task coverage relative to full context for Qwen3.5-35B-A3B on SWE-bench (threshold/depth sweeps, 25 tasks).

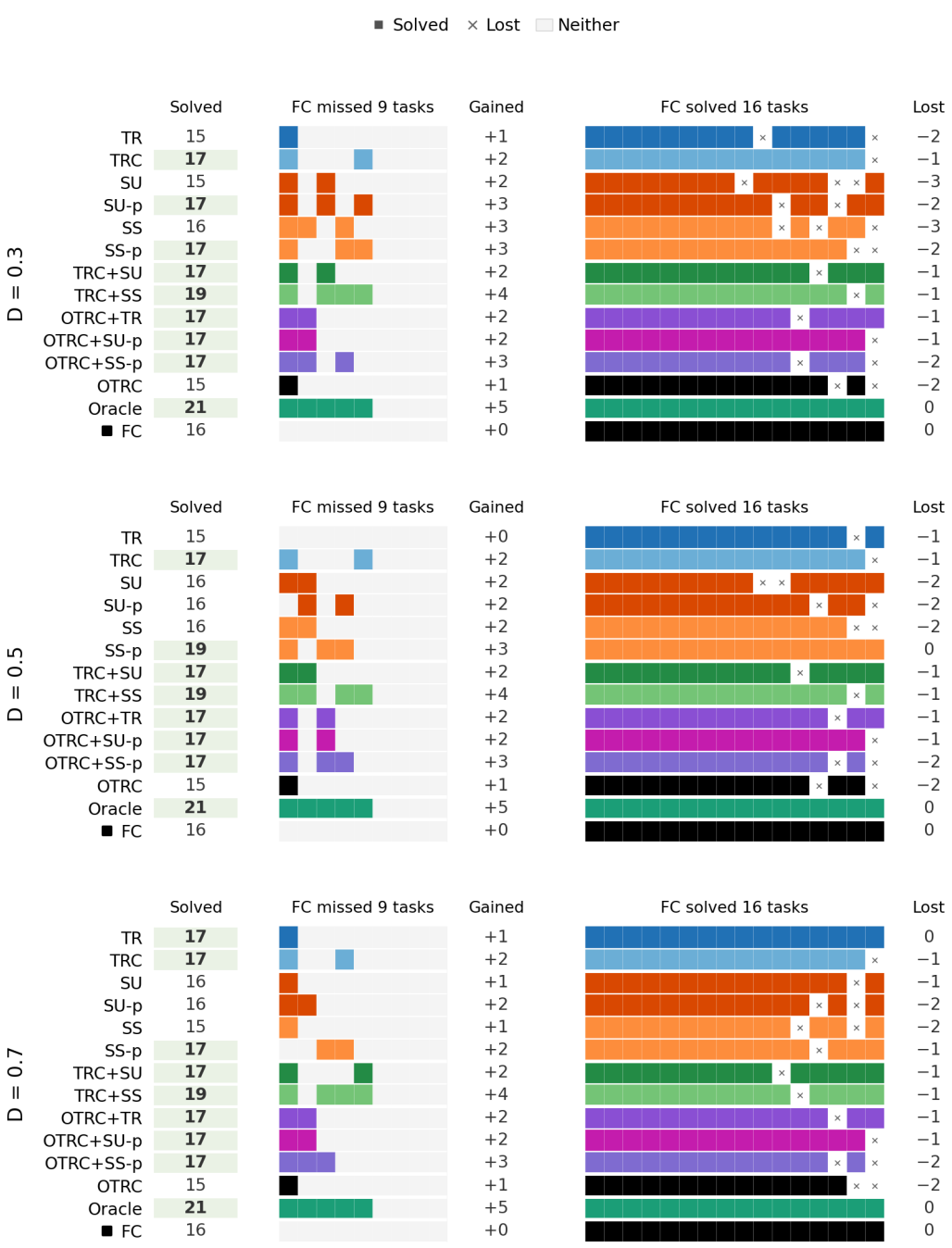


Figure 12: **Loose threshold (20K).** Task coverage relative to full context for Qwen3.5-35B-A3B on SWE-bench (threshold/depth sweeps, 25 tasks).

A.6.3 SUMMARIZER MODEL SENSITIVITY

We examine whether changing the summarizer affects task success and execution efficiency while keeping the agent fixed. Figure 13 compares self-summarization with Qwen3.5-9B and Gemma-4-12B for SU and TRC+SU on the 25-task SWE-bench subset.

SU is more sensitive to the summarizer model: resolve rate falls from 53.3% with self-summarization to 37.3% with Qwen3.5-9B and 28.0% with Gemma-4-12B. TRC+SU remains closer to its 68.0% baseline, achieving 62.7% and 66.7%, respectively. Neither alternative reduces mean

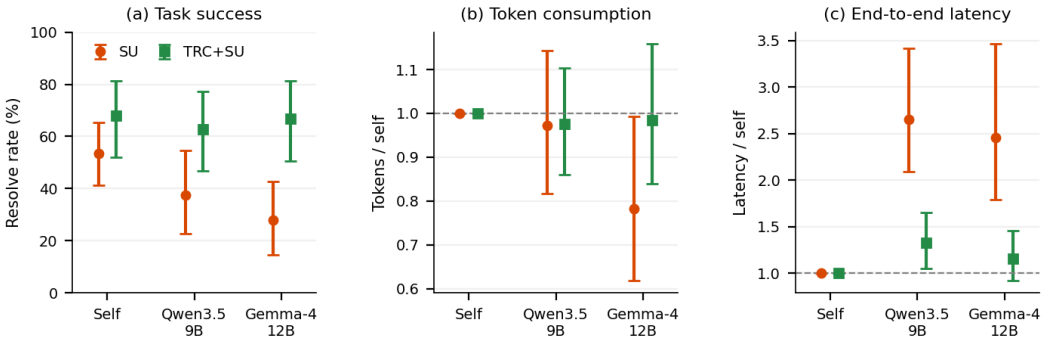


Figure 13: Summarizer sensitivity with a fixed Qwen agent on SWE-bench. Success uses 25 tasks; resource ratios use 10 shared tasks for SU and 23 for TRC+SU, relative to self-summarization. Bars show 95% task-bootstrap intervals.

end-to-end latency on the shared eligible tasks. These results suggest that smaller summarizers do not automatically improve efficiency, and their effect on task success depends on the compression policy.

A.6.4 Run outcomes and resource limits

Figure 14 distinguishes successful execution from different unresolved outcomes at the primary configurations. Each bar includes all three attempts per task: 300 on SWE-bench and 120 on Terminal-Bench. Successful verdicts take priority; remaining attempts are classified as time-limited, step-limited, submitted but unsuccessful, or other/missing outcomes. The breakdown reveals different failure patterns. On SWE-bench, GLM's SU and SS runs frequently reach the time limit, whereas Devstral's step-triggered policies frequently reach the step limit. On Terminal-Bench, time limits account for a substantial share of unresolved attempts across models. These patterns provide context for resource comparisons that exclude time-capped runs.

A.6.5 Changes in execution length

Figure 15 compares agent call counts with full context on tasks solved in at least two of three attempts by both policies. We average calls over successful attempts within each task and report the percentage change relative to FC. Summarizer calls are excluded.

Changes are generally smaller on SWE-bench than on Terminal-Bench. On Terminal-Bench, OTRC+TR increases mean calls by approximately 180% for Qwen and 152% for Devstral. Other policies show smaller increases or decreases, indicating that compression does not uniformly lengthen execution. These comparisons describe shared solved tasks rather than the full cohort; Terminal-Bench estimates use only 2–15 tasks and should be interpreted alongside their uncertainty intervals.

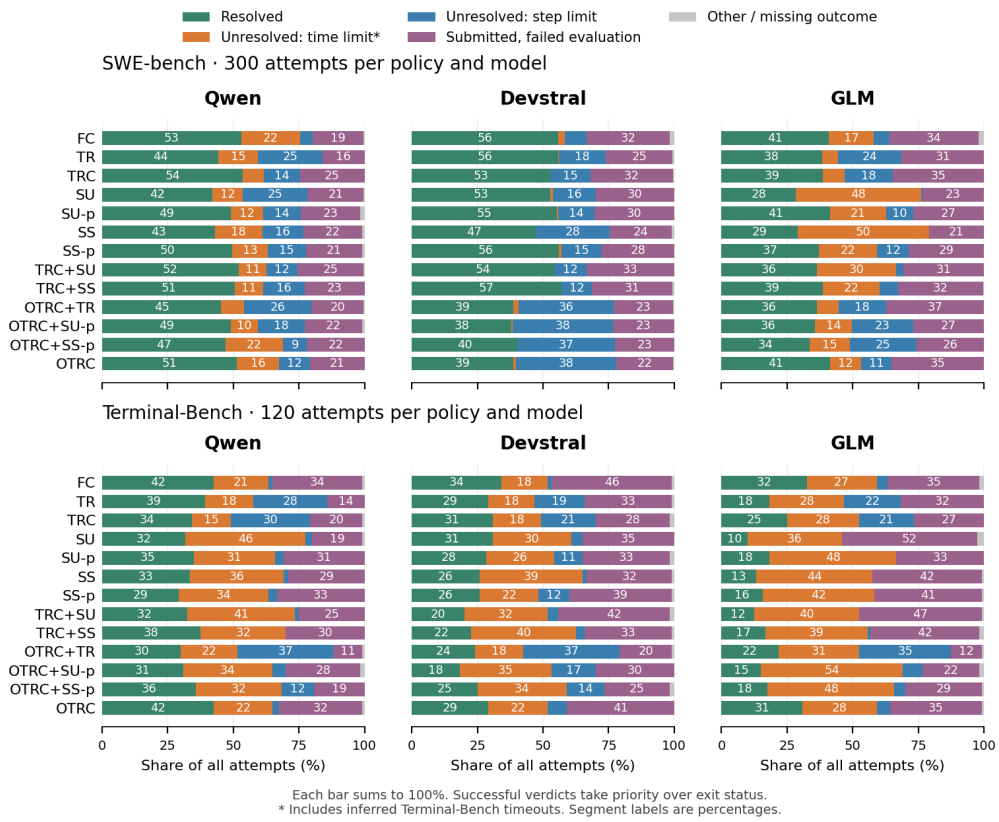


Figure 14: Run outcomes at the primary configurations. Each bar sums to 100% of attempts, with successful verdicts taking priority over exit status. Time-limit categories include inferred Terminal-Bench timeouts; labels show percentages.

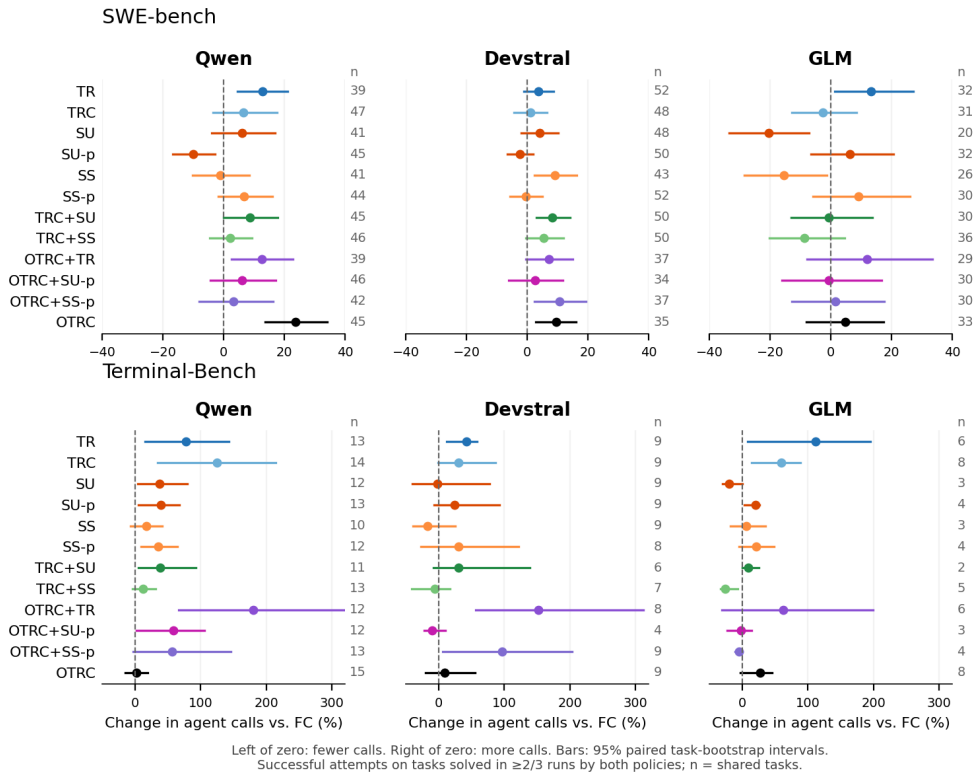


Figure 15: Change in agent calls relative to FC on shared solved tasks. Positive values indicate more calls; bars show 95% paired task-bootstrap intervals, and n denotes shared tasks. Axis scales differ between benchmarks.