SLIPSTREAM: TRAJECTORY-GROUNDED COMPACTION VALIDATION FOR LONG-HORIZON AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

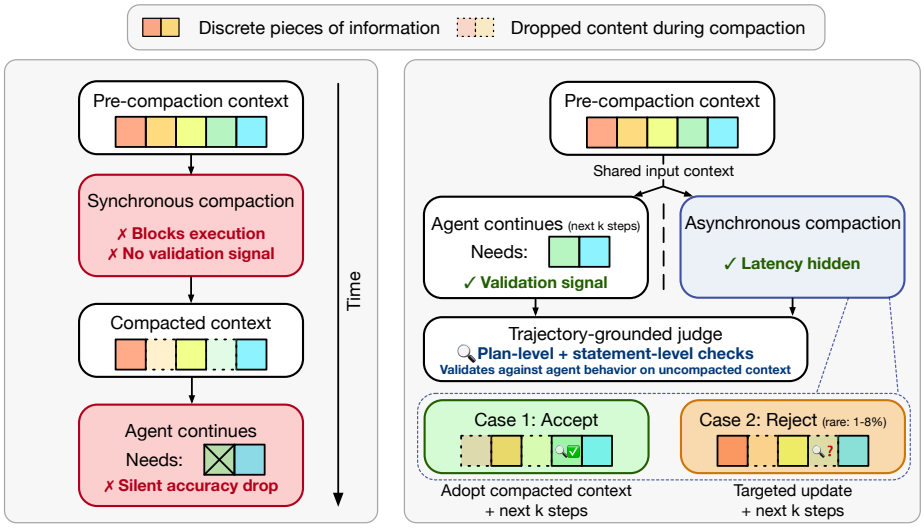
To cope with the large contexts that long-horizon LLM agents produce, modern frameworks increasingly rely on compaction – invoking an LLM to rewrite the accumulated trajectory into a shorter summary that the agent resumes from. Today, compaction runs synchronously on the critical path of agent execution but this can unpredictably degrade accuracy due to a structural validation gap: the compactor must condense context but is fundamentally unaware of precisely what information the agent will need later. Further, because post-compaction agent steps are conditioned on the new summary, targeted validation criteria do not exist and errors silently propagate through coherent but incorrect behavior. Our key insight is that asynchronous compaction efficiently addresses this gap: by running the compactor in parallel with continued agent execution on the original context, the candidate summary and the agent's next steps are generated independently from the same pre-compaction state, yielding a validation signal independent of the summary itself. We build *Slipstream*, a trajectory-grounded compaction system that uses a judge to validate the candidate summary against the agent's continued reasoning, checking that it preserves both the agent's forward intent and the key facts and constraints it depends on. Across long-horizon coding (SWE-bench Verified) and web-browsing (BrowseComp) workloads, Slipstream improves task accuracy by up to **8.8 percentage points** while reducing end-to-end latency by up to **39.7%**.

1 INTRODUCTION

Long-horizon LLM agents have rapidly become a dominant paradigm for solving complex tasks that require sustained reasoning over many rounds of tool use, observation, and planning (Yao et al., 2023; Yang et al., 2024; Schick et al., 2023; Nakano et al., 2021). As these agents tackle longer horizons – from multi-step software engineering to extended web research – their context windows accumulate observations, intermediate results, and partial progress that may become relevant many steps later. This growth quickly degrades model behavior: accuracy drops with context length even when all relevant information remains present, a phenomenon known as context rot with reported accuracy drops of 14–85% (Liu et al., 2024; Hong et al., 2025; Du et al., 2025). To manage this, modern agent frameworks have converged on *compaction*: invoking an LLM to rewrite the accumulated trajectory into a shorter summary that the agent then resumes from directly (Anthropic, 2026b; Microsoft, 2026; LangChain, 2025). Compaction is now triggered routinely throughout long-horizon execution rather than as a last resort at the model's context limit – Anthropic, for example, recommends compacting at as little as 5–20k tokens for some workloads (Anthropic, 2026a).

As deployed today, compaction runs *synchronously* on the critical path of agent execution, pausing the agent until the compactor produces a summary of the context. Beyond the direct latency overheads that this introduces – accounting for 26–44% of end-to-end agent execution time across mainstream workloads – the deeper issue is one of accuracy. Compaction must decide what to preserve without knowing what the agent will need next, and synchronous execution provides no way to validate this decision: once the summary replaces the original context, every subsequent step is conditioned on the summary itself, so the agent's continued behavior cannot serve as an independent check. When the compactor drops or distorts information the agent later needs, task outcomes silently degrade, with no error signal at the moment of compaction. Traditional summarization metrics do not address this since they operate only on the current context and the candidate summary, not on the future agent behavior that determines whether the summary is sufficient. What is missing is a validation signal grounded in the agent's continued trajectory itself.

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.



(a) Synchronous compaction (today) (b) **Slipstream**: trajectory-grounded validation

Figure 1: Synchronous compaction vs. *Slipstream*. (a) Synchronous compaction blocks agent execution and offers no visibility into what future actions require, leading to silent accuracy degradation. (b) *Slipstream* runs the compactor in parallel with continued agent execution on the original context, hiding compaction latency. The next-k agent actions provide a held-out validation signal covering intent and facts for the trajectory-grounded judge, which either accepts the candidate summary for adoption or triggers a targeted update before adoption.

Our key insight is that *asynchronous compaction* is the only execution mode that produces this missing validation signal without duplicating the agent's execution. When the compactor runs in parallel with the agent's continued execution on the original, uncompacted context, the agent's next steps and the candidate summary are generated independently from the same pre-compaction input. This independence is crucial: in synchronous execution, any agent behavior used for validation is already conditioned on the summary it is meant to check, removing its value as an independent signal. Asynchrony is therefore not merely a way to remove blocking latency – it is what makes trajectory-grounded validation possible.

Based on this insight, we present ***Slipstream***, a trajectory-grounded compaction system for long-horizon agents that runs the compactor in parallel with continued agent execution on the uncompacted context (Figure 1). Once the compactor returns, Slipstream applies a judge to validate the candidate summary against the next-k agent steps that completed during compaction. The judge performs two complementary checks, exploiting the structure of agentic execution: a *plan-level* check that the summary preserves the agent's forward intent, and a *statement-level* check that the specific facts, constraints, and intermediate results the agent relies on are preserved. On acceptance (the predominant case), execution resumes from the compacted state augmented with the already-generated next-k steps; otherwise, Slipstream performs a targeted update to the summary using the judge's diagnosis. Two properties make this design feasible. First, compaction errors that affect downstream behavior most often surface within the asynchronous window; across our workloads, 88-100% of first error manifestations arise within 3 agent steps. Second, long-horizon agent serving is memory-constrained, leaving headroom to run the compactor and judge alongside the main agent at low cost.

Across long-horizon coding (SWE-bench Verified) and web-browsing (BrowseComp) workloads with Qwen3.5-9B and Seed-OSS-36B-Instruct, Slipstream improves task accuracy by up to **8.8 percentage points** over synchronous compaction while reducing end-to-end latency by up to **39.7%**. Both wins stem from the same mechanism: asynchrony moves compaction off the critical path, and provides the independent behavioral signal that exposes summary errors synchronous baselines silently adopt.

2 Background and Related Work

Modern agents solve long-horizon tasks by iterating through a reason-then-act loop: they reason over the accumulated conversation, issue tool calls, observe results, and append each step back into the context (Yao et al., 2023; Yang et al., 2024; Schick et al., 2023; Nakano et al., 2021). This context is the agent's working state, recording observations, constraints, partial progress, and unresolved

subgoals. The rate of context growth varies by agent paradigm – single-agent strategies accumulate the full trajectory, while task-decomposition approaches (Sun et al., 2026; Kim et al., 2024a) split work across sub-agents that return shorter results. Regardless, as task horizons grow, context grows until the agent either reaches the model's maximum context length or suffers degraded performance before that limit. This degradation is not merely a capacity issue: LLM accuracy drops with context length even when all relevant information remains present, a phenomenon known as context rot (Liu et al., 2024; Hong et al., 2025; Du et al., 2025), with reported degradations of 13.9–85 % (Du et al., 2025). Long-horizon agents thus require active *context management* well before the nominal window is exhausted.

The design space of context management spans three broad classes. First, *KV-cache compression* operates at the inference level, evicting or compressing key-value entries to reduce memory and bandwidth costs while preserving the underlying token sequence (Li et al., 2024; Cai et al., 2025; Behnam et al., 2025). These techniques target the computational cost of long contexts but do not change what the model attends to logically, as the agent's working state retains the full trajectory. Second, *retrieval-augmented and external-memory approaches* offload accumulated history to an external store and fetch relevant content on demand, ranging from semantic-similarity retrieval over chunked logs (Lewis et al., 2020) to memory-augmented architectures that maintain evolving structured state (Packer et al., 2023; Wang et al., 2026; Xu et al., 2025). These methods preserve information losslessly in storage but introduce retrieval-time decisions about what to surface, which can be brittle with noisy or similar fragments (Wang et al., 2026). Third, *context compaction* rewrites the in-context working state into a shorter representation that the agent continues from directly (Wu et al., 2025; Lu et al., 2025; Anthropic, 2026c). Compaction differs from the first two in that it modifies the agent's logical context rather than its physical representation or its access path; the agent operates on the compacted state as if it were the trajectory itself.

Compaction has emerged as the dominant approach for long-horizon agents in recent agent frameworks (Anthropic, 2026c; LangChain, 2025). Earlier heuristic pruning schemes (Yang et al., 2024; Packer et al., 2023; Wang et al., 2025) – e.g., dropping oldest messages, obsolete tool outputs, or intermediate reasoning traces – assumed that removable content can be identified by shallow signals like recency or message type, an assumption that is brittle for long-horizon tasks where seemingly incidental information may later become necessary. Compaction, by contrast, uses an LLM to produce a holistic summary that can in principle preserve information based on its semantic relevance rather than its position in the trace. In practice, agent systems trigger compaction aggressively, well before the context is full, to stay ahead of context rot. Anthropic, for example, recommends not waiting until the context is full: compaction should be triggered at roughly 5–20k tokens for some workloads and 50–100k tokens for more complex tasks (Anthropic, 2026a). Thus, compaction is now a routine part of long-horizon execution rather than a last-resort recovery mechanism at the context limit.

Today, compaction is typically performed *synchronously*. When triggered, the agent pauses, a compactor LLM generates a summary, the summary replaces the original context, and agent execution resumes. This design is natural for autoregressive generation; the next agent step is conditioned on the current context, so the system updates the context before resuming.

3 MOTIVATION

Synchronous compaction has two costs that limit its effectiveness in practice. It can silently degrade accuracy when the compactor drops or distorts information the agent later needs, and it adds substantial latency by blocking task execution while the compactor runs. We examine each in turn.

Silent deviation with unpredictable accuracy impact. Compaction is lossy by design – it must decide what to preserve, what to compress, and what to drop. But at compaction time, there is no way to know what information the agent will need later; indeed, discovering the relevant future trajectory is precisely what we employ the agent to autonomously determine. Compaction is therefore inherently a prediction about future relevance made without access to future behavior, and when that prediction is wrong, the summary may omit facts the agent later needs, collapse important intermediate progress into a vague statement, or introduce information the original context did not support.

Figure 2 provides real examples of such failures from popular coding and web-browsing agents. With omission errors, the summary drops information the agent later needs, e.g., in Case A, the summary silently drops the correct candidate from a set of three, and the agent never recovers it, producing

Case A: Candidate inversion [*Omission*]
Task: Identify a first name shared by two people from different industries whose mothers also share a first name.

① **Pre-compaction context:** `Tom is partially verified; Declan and Brian remain unverified candidates.`

② **Corrupted compaction:**
"Awaiting verification: Declan and Brian"
The (ultimately) correct answer, 'Tom', was evicted.

Correct continuation (on original context):
"Continue verifying Tom."

③ **Post-compaction failure:** `Investigates Declan next; Tom is never verified, leading to an error.`

Case B: Intent mutation [*Commission*]
Task: Fix a bug by removing calls to a function from `a.py`, while keeping its calls in `b.py`.

① **Pre-compaction context:** `Finished editing a.py; b.py should remain untouched.`

② **Corrupted compaction:**
"Remove the function from everywhere."
Scoped edit became a global one.

Correct continuation (on original context):
" All necessary edits in a.py are done; leave b.py as is and start testing."

③ **Post-compaction failure:** `Removes the function everywhere; code breaks and tests fail.`

Case C: Entity replacement [*Omission + commission*]
Task: Find a publication featuring sideshow performers, including one who was 18 inches tall.

① **Pre-compaction context:** `The 18-inch performer is identified as Tom Thumb; the publication remains to be found.`

② **Corrupted compaction:**
" Jeffrey Hudson is a 18" sideshow performer."
Correct entity replaced by a fabricated one.

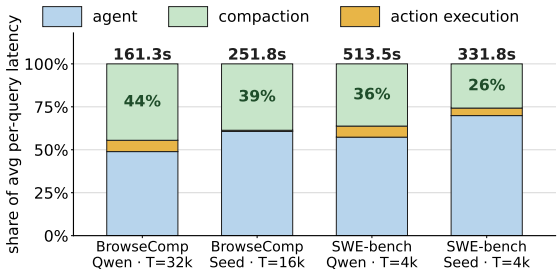
Correct continuation (on original context):
"The 18-inch performer is Tom Thumb. Search for publications."

③ **Post-compaction failure:** `Searches for the wrong entity; the original trail of clues is lost and never recovered.`

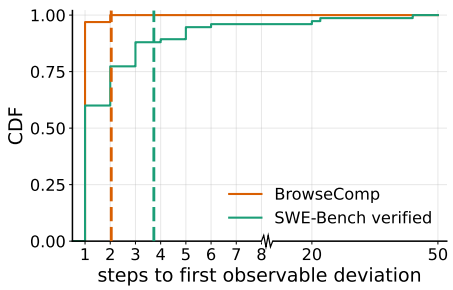
Figure 2: Three illustrative compaction failures from real agent traces on browsing and coding workloads. Each case shows ① the critical context *before* compaction, ② the corrupted compacted state, and ③ the resulting failure behavior. Dashed = correct continuation (counterfactual; not produced under synchronous compaction); green = correct/preserved information; red = corrupted or injected error.

a wrong final answer. With commission errors, the summary introduces or mutates information not supported by the original context, e.g., in Case B, a targeted patch is distorted into a blanket instruction to remove invocations of a function everywhere, leading the agent to make changes the user did not request. Some failures combine both forms, as in Case C, where the summary drops the confirmed entity ("Tom Thumb") and replaces it with a fabricated one, sending the agent down the wrong search path that never recovers the missing information. Across our workloads, omission accounts for roughly 90% of failures, with commission and combined failures making up the remainder.

These are not separate pathologies but manifestations of the same underlying problem: the compactor's prediction about what would matter for the agent was incorrect, and synchronous compaction has no mechanism to validate that prediction before committing to it. Put differently, once the summary replaces the original context, the handoff is *irreversible* – every subsequent step is conditioned on the summary itself, so the agent's continued behavior cannot serve as an independent check. A corrupted summary produces steps consistent with the corruption rather than with the original trajectory, leaving the system without an immediate error signal. The failure is therefore silent, and the accuracy damage only becomes visible when task outcomes degrade.



(a) Normalized latency breakdown of synchronous compaction. Each workload runs with either Qwen3.5-9B or Seed-OSS-36B-Instruct as the agent, with different compaction thresholds (**T**).



(b) Locality of compaction-induced silent deviation.

Figure 3: Challenges with synchronous compaction. (a) Compaction lies on the agent's critical path, accounting for 26–44% of end-to-end latency despite producing no task-useful action. (b) Distribution of agent steps between compaction and the first observable trajectory deviation due to compaction errors; most deviations appear within the first few steps, well within Slipstream's typical async window (dashed lines).

Existing metrics for traditional summarization fail to address this gap. Source-grounded metrics such as faithfulness (Kryscinski et al., 2020), coverage (Nenkova & Passonneau, 2004), QA-based probes (Durmus et al., 2020), and LLM-as-judge scoring (Liu et al., 2023) evaluate the summary against the original context, but they treat all source facts as equally important and correlate poorly with human judgments of content importance (Deutsch et al., 2022; Scialom et al., 2021). For agent compaction, this manifests as a fundamental tradeoff: reliably preserving the facts the agent *will* need requires near-complete preservation of the source (defeating the purpose of compaction), while any relaxation risks dropping precisely what the agent later needs. The metric has no principled way to prioritize, because facts matter only in relation to *future agent behavior*, which it cannot see. This is particularly damaging since omissions dominate the error distribution but produce no contradiction with the source for these metrics to detect (Zhang et al., 2023; Kim et al., 2024b). Reference-grounded metrics like ROUGE (Lin, 2004) are inapplicable since compaction has no reference summary.

Taken together, what existing metrics miss is *sufficiency*: a compacted state must preserve enough of the original context for the continuing agent to avoid deviating from the trajectory it would have followed without compaction. Sufficiency depends on future agent behavior, which is precisely what source-grounded validation cannot see. This motivates a new validation signal – one grounded in the agent behavior that compacted state is meant to support, rather than in the source context it replaces.

Blocking execution and high latencies. Beyond its accuracy risk, synchronous compaction stalls task progress. While the compactor LLM runs, the agent cannot issue tool calls, perform reasoning, or make progress on the task. Each compaction event requires an LLM inference call but produces no task-useful actions. Moreover, this overhead compounds over long-horizon tasks, especially since modern systems trigger compaction aggressively to stay ahead of context rot rather than waiting until the context window fills. Figure 3a breaks down the (normalized) end-to-end latencies when using synchronous compaction at varying frequencies across our agent workloads. As shown, compaction accounts for an average of 26-44 % of total per-query latency across the considered settings.

4 METHOD

Slipstream turns compaction into an asynchronous validation procedure: when compaction is triggered, the agent continues executing on the original, uncompacted context while a compactor runs in parallel on a separate thread. Because both processes operate from the same pre-compaction input, the candidate summary and the agent's continued behavior are generated independently, providing the held-out validation signal missing under synchronous execution (Section 3). Slipstream uses the same context-length trigger as the synchronous baseline, so any performance difference comes from how compaction is scheduled and validated rather than when it occurs. We refer to the agent's continued execution during compaction (i.e., its tool calls and reasoning steps) as the *next-k trajectory*. Once the compactor returns, a trajectory-grounded judge analyzes the summary against the next-k trajectory and either accepts the candidate for adoption or returns a diagnosis used to update it before adoption. Figure 1 illustrates this execution model and contrasts it with the synchronous baseline.

4.1 WHY A FINITE TRAJECTORY WINDOW SUFFICES

Slipstream observes only a finite continuation of the agent's execution – the next-k steps generated while compaction runs. The effective k is not a fixed parameter but is instead naturally determined by each compaction duration relative to agent step time. Compaction is reliably longer than a single agent step because it generates a substantial summary, while agent steps typically generate only short reasoning traces and tool calls. In our evaluations, compaction overlaps with an average of 2.1 agent steps on BrowseComp and 3.7 steps on SWE-bench Verified.

To assess whether this window is sufficient for validation, we measure how quickly compaction errors manifest in agent behavior (Figure 3b). On BrowseComp, where search-style tasks tightly couple context to action, 97% of first error manifestations occur at $k = 1$ and 100% by $k = 2$, well within Slipstream's average window. In contrast, for SWE-bench Verified where coding tasks involve intermediate reasoning chains before actions commit to corrupted information, the distribution is broader: 60% of errors at $k = 1$, 88% by $k = 3$, and 96% by $k = 6$. Here, windows with Slipstream cover 88% of first error manifestations; the few errors that surface outside of Slipstream's validation window (and thus, only after the compaction summary is applied) cannot be recovered, as with synchronous compaction. Overall, both distributions exhibit a *locality property*: the first observable error manifests soon after compaction, within the agent's current subgoal, allowing the (finite) asynchronous window to capture them.

4.2 VALIDATING SUMMARIES WITH THE TRAJECTORY-GROUNDED JUDGE

Slipstream's judge decides whether the candidate compacted state is acceptable, decomposing this into two complementary checks against the next-k trajectory: one verifies that the compacted state is consistent with the agent's downstream intentions, and the other verifies that it preserves the specific information those steps rely on. The reason is that these two properties can fail independently – a summary can preserve every relevant fact while changing the agent's forward intent, or preserve intent while losing the facts needed to act on it. We describe each check below:

- The *statement-level check* asks whether the compacted state preserves the concrete facts, constraints, intermediate results, and tool observations explicitly used in the next-k trajectory. The agent's thinking tokens reveal not only the facts driving its immediate actions, but also facts it previously relied on along the current trajectory. This check targets content errors exemplified in Figure 2, such as the omissions in Case A and the entity-level corruptions in Case C.
- The *plan-level check* asks whether the compacted state supports the same forward intent as the original continuation. This targets structural drift: the summary may preserve isolated facts while steering the agent toward a different next action, as in Case B. The check is feasible because long-horizon agents articulate their plans explicitly to stay coherent across many steps. They typically do this in one of two ways: through coarse end-to-end framing at task start that is periodically refined with shorter-term plans as the agent acts ("I will first locate the affected imports, then modify each file") as in SWE-Bench and GAIA (Mialon et al., 2024), or through an explicit (global) running checklist that updates with progress as in BrowseComp. The next-k trajectory reveals the current plan, enabling direct validation on articulated intents. If plans are revised within the next-k window, the judge validates against the most recent articulation.

On accept, the judge passes the candidate to the adoption procedure (§ 4.3); on reject, it returns a brief diagnosis of the missing or corrupted trajectory-relevant information. By default, the judge uses the same model as the agent with a structured prompt (Appendix D.1); Section 5 shows that judge validation is largely insensitive to model choice and contributes materially beyond asynchrony alone.

4.3 ADOPTION AND TARGETED UPDATE

On judge acceptance, Slipstream swaps execution to the compacted state augmented with the next-k continuation steps that completed during compaction. Because the judge has already verified consistency, the live agent can safely continue from the same task progress with a reduced context. On rejection, Slipstream briefly pauses agent execution and performs a targeted update to the compacted state, using the judge's diagnosis and the relevant evidence from the uncompacted continuation to repair the specific trajectory-relevant omission or corruption – e.g., restoring a dropped constraint, correcting a mutated entity, or preserving the latest patch intent – without running synchronous compaction from scratch. Slipstream then adopts the updated compacted state together with the

Table 1: **Slipstream**'s absolute accuracy gain over Sync across models, workloads, and compaction thresholds. Thresh. denotes the active-context token count at which compaction is triggered. Slipstream improves accuracy in every configuration (by up to 8.8%). Multi-run results with significance tests are in Appendix B.

	SWE-bench verified			BrowseComp		
Model	**Thresh.**	**Sync**	**Slipstream**	**Thresh.**	**Sync**	**Slipstream**
Qwen3.5-9B	4k	23.4%	**29.8%** (+6.4%)	32k	53.3%	**57.3%** (+4.0%)
	6k	22.0%	**30.8%** (+8.8%)	48k	52.7%	**57.3%** (+4.6%)
	8k	20.4%	**29.2%** (+8.8%)	64k	55.3%	**58.0%** (+2.7%)
Seed-OSS-36B	4k	29.6%	**35.8%** (+6.2%)	16k	46.7%	**48.7%** (+2.0%)
	6k	30.6%	**37.4%** (+6.8%)	24k	52.0%	**53.3%** (+1.3%)
	8k	38.0%	**40.6%** (+2.6%)	32k	49.3%	**52.7%** (+3.4%)

next-k continuation steps. Empirically, rejection is rare (1.0–3.5% on BrowseComp, 5.4–8.5% on SWE-bench Verified), so Slipstream retains most of the latency benefit while using targeted updates as a guardrail against summaries the judge identified as trajectory-changing. When a targeted update cannot recover (e.g., structural drift too pervasive to repair locally), Slipstream falls back to synchronous compaction.

4.4 System considerations

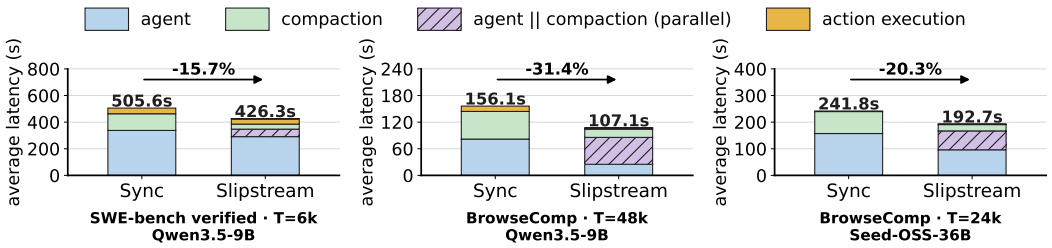
Slipstream exploits structural properties of long-horizon agent serving to make asynchronous execution practical. The key enabler is that modern long-horizon agent serving is constrained by memory rather than compute (Wu et al., 2026; Kwon et al., 2023): effective batch sizes sit well below the available compute limit, leaving headroom to run the compactor and judge alongside the main agent at minimal additional cost. Within this headroom, Slipstream further optimizes its use of memory and compute. The agent thread and the compaction thread operate over the same pre-compaction context, so Slipstream reuses the shared prefix in the KV cache instead of materializing two independent copies, similar to prior work on shared-prefix inference (Pan et al., 2025). To avoid duplicating memory loads over this prefix, we adopt multi-layer cascade attention (Ye et al., 2024), which loads the prefix once and lets both threads attend to it without redundant memory traffic. The judge's inputs are also lightweight: it processes only the candidate compacted state and the next-k trajectory (typically 3.6-6.8% of the pre-compaction context size) and outputs a brief accept/reject decision with a diagnosis on rejection.

5 Experiments

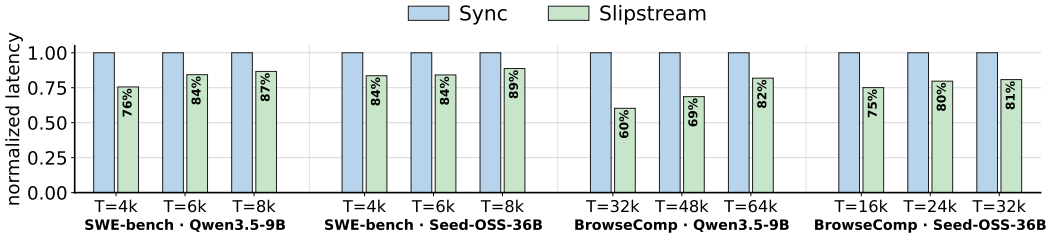
5.1 Setup

Models and workloads. We evaluate Slipstream using two models instruction-tuned for long-horizon agentic tasks: **Qwen3.5-9B** (Qwen Team, 2026), a hybrid-attention model, and **Seed-OSS-36B-Instruct** (ByteDance Seed Team, 2025), a dense Transformer. We evaluate on two representative long-horizon agent workloads. **BrowseComp** (Wei et al., 2025) evaluates web-browsing agents that must maintain search progress and entity constraints across many tool calls. **SWE-bench verified** (Jimenez et al., 2024) evaluates coding agents that must preserve issue context, test outcomes, and repository observations over multi-step debugging trajectories. These workloads stress different forms of trajectory state: BrowseComp primarily requires search-state maintenance, while SWE-bench requires preserving both factual observations and evolving code-editing plans.

Baselines, methodology, and hardware. We compare Slipstream with a synchronous compaction baseline (**Sync**). Both schemes invoke compaction when the active context reaches the same threshold, run in the same harness – mini-swe-agent (SWE-agent Team, 2025) (linear history, bash-only) for SWE-bench verified and the released FoldAgent code (Sun et al., 2026) for BrowseComp – and, unless otherwise stated, share the agent and compactor prompts, tool interface, and summary-length budget. They differ only in that Sync blocks agent execution and always adopts the compacted context, whereas Slipstream compacts asynchronously and validates or updates the candidate using the next-k trajectory (Figure 1). For each model-workload pair, we evaluate three compaction thresholds: a default threshold set to approximately $\sim$20% of the average full trajectory length, plus thresholds at $\frac{2}{3}\times$ and $\frac{4}{3}\times$ the default to assess sensitivity to the compaction frequency. In all evaluations, the



(a) Per-query latency breakdown for three representative configurations, decomposing total runtime into agent reasoning, action execution, synchronous compaction (Sync only), and overlapped agent–compaction execution (Slipstream only). We omit judge/update overhead as it is negligible ($< 1\%$ of total latency across evaluations).



(b) Normalized end-to-end latency (Slipstream / Sync) across all evaluated configurations.

Figure 4: Latency analysis across workloads, models, and compaction thresholds (**T**). Slipstream consistently reduces latency by 11.3-39.7% via moving compaction off the critical path, with greater reductions at smaller thresholds, where compaction is invoked more frequently. Full results are in Appendix A.

judge outputs a 0–10 score with brief reasoning; scores below 7 trigger the targeted update path (Section 4.3). We report task success rate as accuracy, and report average per-query latency broken down into agent reasoning, action execution, and compaction, omitting the negligible judge/update overhead ($< 1\%$ of total latency). We serve all models locally with vLLM 0.19.1; `Qwen3.5-9B` runs on one NVIDIA H100 80GB GPU, while `Seed-OSS-36B-Instruct` runs on two.

5.2 Main Results

Accuracy. Table 1 shows that Slipstream improves task success rate over synchronous compaction in every configuration. On BrowseComp, Slipstream improves accuracy by 1.3–4.6%, suggesting that it better preserves active search state and avoids silent repetitions or unintended changes in search direction after compaction. On SWE-bench verified, the gains are larger (2.6–8.8%), indicating that trajectory-grounded validation is especially useful for preserving patch intent, test interpretation, and repository observations that synchronous compaction may distort or omit.

Latency. Figure 4a shows detailed latency breakdowns for three representative workload–model combinations at their default compaction thresholds (breakdown results for all settings are in Appendix A). In the Sync baseline, compaction appears as a blocking component on the critical path. In contrast, Slipstream overlaps compaction with continued agent execution on the original context, so most of the compaction time is hidden within the asynchronous execution window and contributes little net latency overhead. Figure 4b summarizes the end-to-end effect across all evaluated settings: Slipstream consistently reduces average per-query latency relative to Sync. Slipstream's relative latency reduction grows as compaction is invoked more frequently, since blocking summarization then accounts for a larger share of end-to-end runtime. The judge is also cheap: on SWE-bench verified with Qwen3.5-9B, judge calls account for only 2.0% of Slipstream's total tokens, and Slipstream still uses 11.4% fewer tokens than Sync overall (Appendix C).

Overall, the results show that asynchrony is not merely an efficiency optimization. By continuing on the original context while compaction runs, Slipstream obtains a held-out trajectory signal that can be used to validate and improve the compacted state before adoption, yielding consistent accuracy gains while also hiding summarization latency. The absence of a monotonic accuracy trend across thresholds reflects a known tension: more frequent compaction better mitigates context rot but also creates more opportunities for silent deviation (Section 3). Slipstream's per-compaction validation makes it robust to this tradeoff.

5.3 MICROBENCHMARKS

Ablation studies on accuracy. Compared to Sync, Slipstream introduces two changes with potential accuracy implications: it generates the next-k agent actions on the uncompacted context asynchronously, and it validates and optionally repairs the candidate summary before adoption. To isolate which of these drives the accuracy gains, we compare Sync and Slipstream against a third scheme, Async-only, which overlaps compaction with agent execution on the uncompacted context but always accepts the resulting summary without validation. As shown in Figure 5, Async-only captures most of Slipstream's latency benefit over Sync, but its accuracy remains close to Sync across all three thresholds – overlapping execution alone does not recover the accuracy lost to corrupted summaries, since Async-only still adopts every compacted state regardless of whether it preserves trajectory-relevant information. Slipstream achieves the greatest latency reduction while improving accuracy by 6.4-8.8% over Sync, showing that the accuracy gain comes from trajectory-grounded validation rather than from asynchrony alone.

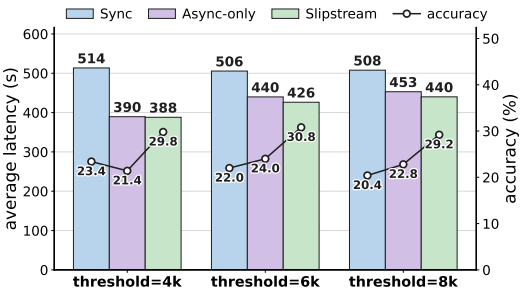


Figure 5: [SWE-bench verified, Qwen3.5-9B] Isolating the contributions of asynchrony and judging. Async-only captures most of Slipstream's latency benefit but does not consistently improve accuracy over Sync, while full Slipstream retains the latency benefit and improves accuracy across all thresholds – indicating that the accuracy gain comes from trajectory-grounded validation rather than asynchrony alone.

Sensitivity to judge model choice. By default, Slipstream uses the same model for the agent and the judge. This is convenient at the system level – both threads share a prefix and a model instance, avoiding redundant prefill during judging and eliminating the need to host a second model (Section 4.4). But it leaves open whether judging actually requires a model as capable as the agent, or whether a smaller, cheaper judge would suffice. To study this, we keep the agent model fixed at Qwen3.5-9B on SWE-bench verified and vary only the judge, swapping in two smaller models from different families. Table 2 shows that these weaker judges consistently outperform synchronous compaction and largely retain the accuracy gains of the same-model judge. This suggests that the gains come from the trajectory-grounded validation signal itself, rather than from relying on an especially strong judge model. It also broadens practical deployment options: the judge can be served by a cheaper model, placed in a separate engine, or scaled independently from the main agent.

Table 2: Judge model ablation on SWE-bench verified with a Qwen3.5-9B agent. Non-Sync rows vary only the judge model. Smaller and cross-family judges consistently outperform Sync and largely retain the accuracy gains of the default same-model judge, indicating that trajectory-grounded validation does not require a judge as capable as the agent.

Threshold	Judge	Accuracy
4k	Sync (no judge)	23.4%
	Qwen3.5-9B	29.8% (+6.4%)
	Qwen3.5-2B	28.2% (+4.8%)
	Llama-3.2-3B	28.2% (+4.8%)
6k	Sync (no judge)	22.0%
	Qwen3.5-9B	30.8% (+8.8%)
	Qwen3.5-2B	28.6% (+6.6%)
	Llama-3.2-3B	31.8% (+9.8%)
8k	Sync (no judge)	20.4%
	Qwen3.5-9B	29.2% (+8.8%)
	Qwen3.5-2B	28.0% (+7.6%)
	Llama-3.2-3B	27.2% (+6.8%)

6 CONCLUSION

We present Slipstream, a trajectory-grounded compaction system for long-horizon LLM agents. Compaction faithfulness has no synchronous ground truth: the compactor must predict what the agent will need next, and synchronous execution offers no way to check that prediction before committing to it. Asynchrony resolves this. Running the compactor in parallel with continued agent execution generates the summary and the agent's behavior independently from the same input, producing a validation signal no synchronous configuration can construct. Slipstream improves task accuracy by up to 8.8 percentage points over synchronous compaction while cutting end-to-end latency by up to 39.7%. More broadly, running auxiliary procedures synchronously on an agent's critical path can foreclose validation signals that asynchrony makes available – a pattern worth examining beyond compaction itself.

AI USE STATEMENT

In this work, we used generative AI tools to help implement methods and to support qualitative and thematic data analysis. We have not used generative AI tools to develop theoretical models or conceptual frameworks, propose or refine hypotheses, design or provide feedback on research methodology or experiments, assist with translation, or interpret results, and the remaining tasks with required disclosure (generating synthetic datasets, formulating or proving mathematical claims, and cleaning or reformatting datasets) are not applicable to this work. Additionally, we used generative AI tools for the following tasks with recommended disclosure: creating and editing software code, creating figures that visualize our experimental results, sourcing and searching for information, identifying relevant literature, drafting parts of the paper, editing the paper to improve readability, and formatting references. We have reviewed all AI-assisted work: AI-assisted code was reviewed and tested for correctness by the authors, AI-assisted qualitative analyses were manually checked by the authors against the underlying agent trajectories, AI-assisted text was checked against our own drafts and experimental records, AI-assisted figures were verified against the underlying experimental data, and all AI-sourced information and literature were verified against the original sources. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI. Separately, LLMs are the object of study in this work: the agents, compactors, and judges we evaluate are LLMs (Section 5); this use is part of the method, not assistance in conducting the research.

REPRODUCIBILITY STATEMENT

We describe the experimental setup, models, and workloads in Section 5. The full judge prompt is provided in Appendix D.1, and the agent prompts for SWE-bench Verified and BrowseComp are provided in Appendix D. To facilitate reproduction, we release the code of Slipstream in an anonymized repository: `https://anonymous.4open.science/r/slipstream`.

REFERENCES

Anthropic. Automatic context compaction. `https://platform.claude.com/cookbook/tool-use-automatic-context-compaction`, 2026a. Claude Cookbook. Accessed: 2026-04-30.

Anthropic. Compaction. `https://platform.claude.com/docs/en/build-with-claude/compaction`, 2026b. Claude API documentation (beta). Accessed: 2026-05-05.

Anthropic. Context engineering: Memory, compaction, and tool clearing. `https://platform.claude.com/cookbook/tool-use-context-engineering-context-engineering-tools`, 2026c. Claude Cookbook. Accessed: 2026-04-30.

Payman Behnam, Yaosheng Fu, Ritchie Zhao, Po-An Tsai, Zhiding Yu, and Alexey Tumanov. RocketKV: Accelerating long-context LLM inference via two-stage KV cache compression. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025.

ByteDance Seed Team. Seed-OSS open-source models. `https://github.com/ByteDance-Seed/seed-oss`, 2025. Accessed: 2026-05-05.

Zefan Cai, Yichi Zhang, Bofei Gao, Yuliang Liu, Yucheng Li, Tianyu Liu, Keming Lu, Wayne Xiong, Yue Dong, Junjie Hu, and Wen Xiao. PyramidKV: Dynamic KV cache compression based on pyramidal information funneling. In *Second Conference on Language Modeling*, 2025.

Daniel Deutsch, Rotem Dror, and Dan Roth. Re-examining system-level correlations of automatic summarization evaluation metrics. In *Proceedings of NAACL*, 2022.

Yufeng Du, Minyang Tian, Srikanth Ronanki, Subendhu Rongali, Sravan Babu Bodapati, Aram Galstyan, Azton Wells, Roy Schwartz, Eliu A. Huerta, and Hao Peng. Context length alone hurts LLM performance despite perfect retrieval. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 23281–23298, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-emnlp.1264. URL `https://aclanthology.org/2025.findings-emnlp.1264/`.

Esin Durmus, He He, and Mona Diab. FEQA: A question answering evaluation framework for faithfulness assessment in abstractive summarization. In *Proceedings of ACL*, 2020.

Kelly Hong, Anton Troynikov, and Jeff Huber. Context rot: How increasing input tokens impacts LLM performance. Technical report, Chroma, July 2025. URL https://trychroma.com/research/context-rot.

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. URL https://arxiv.org/abs/2310.06770.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. In *Proceedings of the 41st International Conference on Machine Learning*, 2024a. URL https://arxiv.org/abs/2312.04511.

Yekyung Kim, Yapei Chang, Marzena Karpinska, Aparna Garimella, Varun Manjunatha, Kyle Lo, Tanya Goyal, and Mohit Iyyer. FABLES: Evaluating faithfulness and content selection in book-length summarization. In *Proceedings of COLM*, 2024b.

Wojciech Kryscinski, Bryan McCann, Caiming Xiong, and Richard Socher. Evaluating the factual consistency of abstractive text summarization. In *Proceedings of EMNLP*, 2020.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023. URL https://arxiv.org/abs/2309.06180.

LangChain. How to manage long context with summarization. https://langchain-ai.github.io/langmem/guides/summarization/, 2025. LangMem documentation. Accessed: 2026-05-05.

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

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. *Advances in Neural Information Processing Systems*, 37:22947–22970, 2024.

Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, 2004.

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. doi: 10.1162/tacl_a_00638.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-Eval: NLG evaluation using GPT-4 with better human alignment. In *Proceedings of EMNLP*, 2023.

Miao Lu, Weiwei Sun, Weihua Du, Zhan Ling, Xuesong Yao, Kang Liu, and Jiecao Chen. Scaling LLM multi-turn RL with end-to-end summarization-based context management. *arXiv preprint arXiv:2510.06727*, 2025.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: A benchmark for general AI assistants. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2311.12983.

Microsoft. Compaction. https://learn.microsoft.com/en-us/agent-framework/agents/conversations/compaction, 2026. Microsoft Agent Framework documentation. Accessed: 2026-05-05.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. WebGPT: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

Ani Nenkova and Rebecca Passonneau. Evaluating content selection in summarization: The pyramid method. In *Proceedings of NAACL-HLT*, 2004.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Rui Pan, Yinwei Dai, Zhihao Zhang, Gabriele Oliaro, Zhihao Jia, and Ravi Netravali. SpecReason: Fast and accurate inference-time compute via speculative reasoning. In *Advances in Neural Information Processing Systems*, 2025.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Thomas Scialom, Paul-Alexis Dray, Sylvain Lamprier, Benjamin Piwowarski, Jacopo Staiano, Alex Wang, and Patrick Gallinari. QuestEval: Summarization asks for fact-based evaluation. In *Proceedings of EMNLP*, 2021.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon agent via context folding. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. doi: 10.48550/arXiv.2510.11967. URL https://arxiv.org/abs/2510.11967.

SWE-agent Team. mini-SWE-agent: The 100 line AI agent that solves GitHub issues. https://github.com/SWE-agent/mini-swe-agent, 2025. Accessed: 2026-09-24.

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 *The Thirteenth International Conference on Learning Representations*, 2025.

Zhenting Wang, Huancheng Chen, Jiayun Wang, and Wei Wei. Memex(RL): Scaling long-horizon LLM agents via indexed experience memory, 2026. URL https://arxiv.org/abs/2603.04257.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. BrowseComp: A simple yet challenging benchmark for browsing agents, 2025. URL https://arxiv.org/abs/2504.12516.

Haoran Wu, Can Xiao, Jiayi Nie, Xuan Guo, Binglei Lou, Jeffrey T. H. Wong, Zhiwen Mo, Cheng Zhang, Przemyslaw Forys, Chengyang Ai, Timi Adeniran, Wayne Luk, Hongxiang Fan, Jianyi Cheng, Timothy M. Jones, Rika Antonova, Robert Mullins, and Aaron Zhao. Combating the memory walls: Optimization pathways for long-context agentic LLM inference. In *Proceedings of the 53rd Annual International Symposium on Computer Architecture*, 2026. URL https://arxiv.org/abs/2509.09505.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, et al. ReSum: Unlocking long-horizon search intelligence via context summarization. *arXiv preprint arXiv:2509.13313*, 2025.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic memory for LLM agents. In *Advances in Neural Information Processing Systems*, 2025.

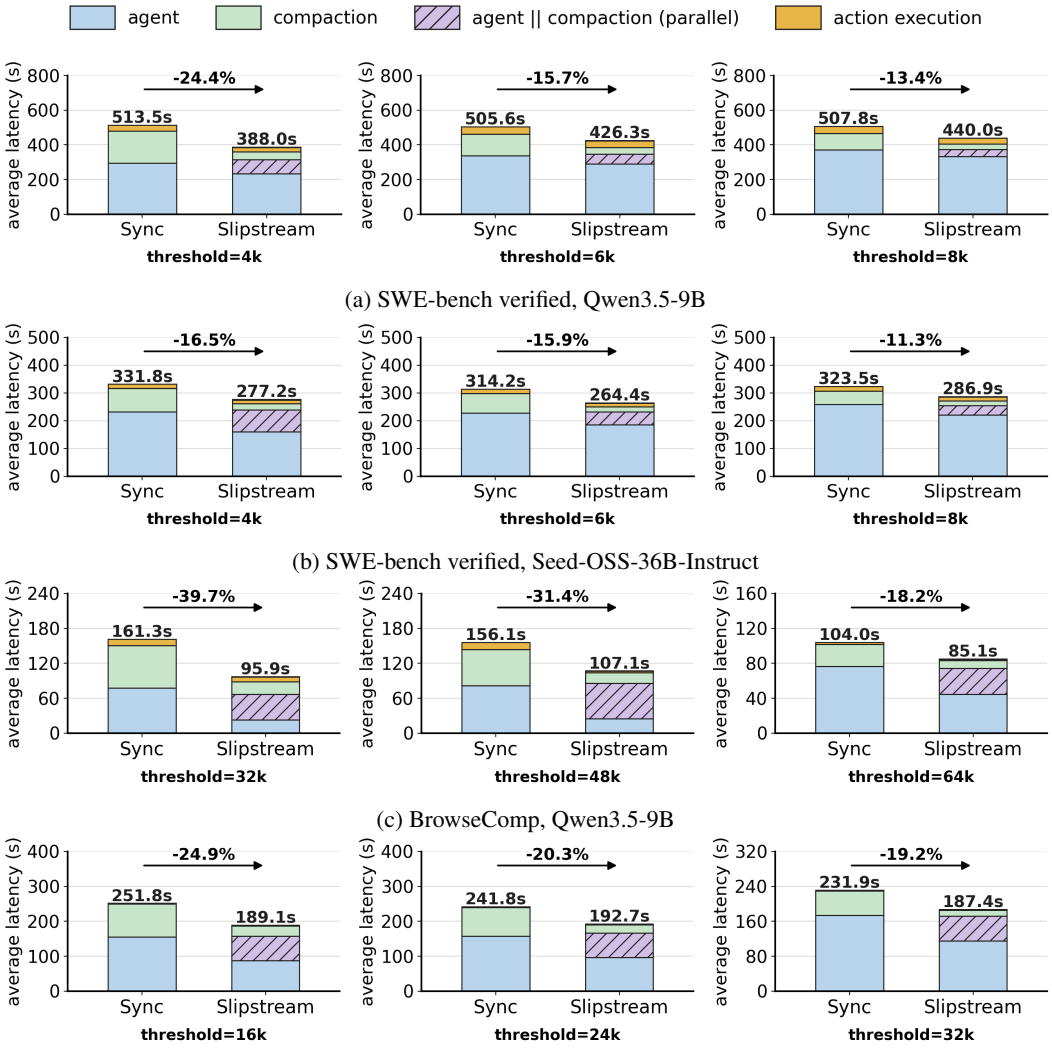
John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

Zihao Ye, Ruihang Lai, Bo-Ru Lu, Chien-Yu Lin, Size Zheng, Lequn Chen, Tianqi Chen, and Luis Ceze. Cascade inference: Memory bandwidth efficient shared prefix batch decoding. `https://flashinfer.ai/2024/02/02/cascade-inference.html`, February 2024. FlashInfer blog post. Accessed: 2026-05-04.

Shiyue Zhang, David Wan, and Mohit Bansal. Extractive is not faithful: An investigation of broad unfaithfulness problems in extractive summarization. In *Proceedings of ACL*, 2023.

A Detailed latency breakdown for all combinations in § 5.2



(a) SWE-bench verified, Qwen3.5-9B

(b) SWE-bench verified, Seed-OSS-36B-Instruct

(c) BrowseComp, Qwen3.5-9B

(d) BrowseComp, Seed-OSS-36B-Instruct

Figure 6: Latency breakdown across workloads, models, and compaction thresholds. Slipstream hides summarization overhead within the asynchronous execution window, achieving near-zero net latency overhead.

B Multi-run comparison and statistical significance

To assess whether the accuracy gains in Table 1 are robust to run-to-run variance, we performed a multi-run comparison between the synchronous baseline and Slipstream. For every model$\times$benchmark combination, we fix one compaction threshold (the first-row setting in Table 1: 4k for SWE-bench verified; 32k and 16k for BrowseComp with Qwen3.5-9B and Seed-OSS-36B, respectively) and run both systems 3 times. We report each system's accuracy as mean $\pm$ standard deviation over runs, together with a one-sided p-value from a pooled paired bootstrap over per-request outcomes, i.e., the probability that Slipstream fails to outperform Sync. Due to limited computational resources, these runs use a fixed 100-request subset sampled once (with a fixed seed) from each benchmark, rather than the full benchmarks used in Table 1; absolute accuracies are therefore not directly comparable across the two tables. Table 3 shows the results. Slipstream improves mean accuracy over Sync in all four settings, with $p < 10^{-5}$ and $p = 0.043$ on SWE-bench verified (Qwen3.5-9B and Seed-OSS-36B, respectively), and $p = 0.069$ and $p = 0.071$ on BrowseComp.

Table 3: Multi-run accuracy (mean $\pm$ std over 3 runs) on a fixed 100-request subset of each benchmark, at the first-row compaction threshold of Table 1. Δ is the difference in mean accuracy; p is the one-sided paired-bootstrap p-value that Slipstream does not outperform Sync.

Benchmark	**Model (Thresh.)**	**Sync**	**Slipstream**	Δ	p
SWE-bench verified	Qwen3.5-9B (4k)	$15.3 \pm 2.5\%$	$\mathbf{26.7 \pm 5.5}\%$	$+11.4\%$	$< 10^{-5}$
	Seed-OSS-36B (4k)	$29.7 \pm 2.1\%$	$\mathbf{34.7 \pm 2.1}\%$	$+5.0\%$	0.043
BrowseComp	Qwen3.5-9B (32k)	$56.3 \pm 0.6\%$	$\mathbf{59.7 \pm 3.1}\%$	$+3.4\%$	0.069
	Seed-OSS-36B (16k)	$43.3 \pm 1.2\%$	$\mathbf{45.7 \pm 0.6}\%$	$+2.4\%$	0.071

C Token cost analysis

Slipstream adds judge calls and occasional re-summaries on top of the compaction work that Sync performs. Table 4 accounts for the total token cost of both schemes on SWE-bench verified with Qwen3.5-9B at $T = 4$k, reporting per-instance means measured from execution logs. Slipstream's numbers include all judge calls and all re-summaries triggered by rejections ($\sim$0.45 per instance).

Table 4: Per-instance token usage (prompt / completion) on SWE-bench verified with Qwen3.5-9B at $T = 4$k. Slipstream's numbers include all judge calls and rejection-triggered re-summaries.

Setting	**Executor**	**Compactor**	**Judge**	**Total**
Sync	641k / 14.2k	172k / 11.1k	–	838k
Slipstream	618k / 13.8k	90k / 5.8k	14.2k / 0.54k	**742k** (-11.4%)

Slipstream uses 11.4% fewer total tokens than Sync, and judge calls account for 2.0% of its total. The reduction comes from shorter trajectories: validated summaries let the agent submit earlier (70 vs. 87.5 steps per episode on average), which reduces both executor and compactor tokens. Because total prompt tokens (722k vs. 813k) and total completion tokens (20.1k vs. 25.3k) are both lower, the per-token cost of an API deployment is also lower. In self-hosted deployments, the parallel compactor and judge run within the memory-bound serving regime and share the KV prefix with the executor (Section 4.4), so they do not compete for the throughput-limiting resource.

D Detailed Prompts

D.1 Trajectory-Grounded Judge Prompt

The judge decides whether a candidate compacted state is sufficient to adopt by checking it against the next-k trajectory—the agent's continued execution on the uncompacted context while compaction ran asynchronously (§ 4.2). It decomposes this into two complementary checks: a *plan-level* check (whether the compacted state supports the same forward intent as the continuation) and a *statement-level* check (whether the compacted state preserves the concrete facts, constraints, and observations the continuation relies on). The prompt returns a structured JSON verdict; on rejection, the diagnosis feeds into the targeted update path (§ 4.3).

```
You are evaluating whether a candidate compacted summary is consistent
with the agent's speculative steps -- the actions the agent took on
the original, uncompacted context while compaction ran asynchronously.

The agent's new context after adoption will be:
  [candidate compacted summary] + [speculative trajectory appended verbatim]

Judge consistency on TWO criteria using ONLY the candidate compacted
summary and the speculative trajectory provided below.

---
CANDIDATE COMPACTED STATE:
{summary_handover}

---
```

```
SPECULATIVE TRAJECTORY (reasoning + tool calls taken during async compaction):
{spec_actions}

---
CRITERIA:

1. PLAN ALIGNMENT
   The compacted state contains open/pending items (marked [PENDING],
   [OPEN], or described as unresolved/unconfirmed). Does the next-k
   trajectory pursue these open items with the same forward intent?
   - 10: Trajectory directly targets the most critical open item(s)
   - 7-9: Trajectory targets an open item, but not the most critical
   - 4-6: Trajectory is topically related but takes an indirect path
   - 1-3: Trajectory addresses something not listed as open/pending
   - 0:   Trajectory is unrelated to any compacted-state content,
          OR compacted state has no open items but trajectory
          continues searching instead of finishing

2. INFORMATION PRESERVATION
   The speculative trajectory's reasoning may reference prior context
   (entities, partial findings, verified facts, tool observations).
   Are those concrete facts preserved in the candidate compacted state?
   - 10: Every entity/fact the trajectory relies on appears in the
         compacted state
   - 7-9: Most referenced information is present; minor details
          missing but reconstructable from context
   - 4-6: Some key information the trajectory depends on is missing,
          which could cause confusion after adoption
   - 1-3: Trajectory relies heavily on information not in the
          compacted state
   - 0:   Trajectory references a completely different task state
          than what the compacted state describes

Respond with JSON only (no markdown, no extra text):
{"plan_alignment": <int 0-10>,
 "information_preservation": <int 0-10>,
 "score": <int 0-10>,
 "reasoning": "<one sentence on the key strength or weakness>"}

The overall score must equal:
  round(0.5 * plan_alignment + 0.5 * information_preservation)
```

D.2 SWE-BENCH VERIFIED WORKFLOW

Our SWE-bench agent runs in the mini-swe-agent harness (SWE-agent Team, 2025), which keeps a linear message history and exposes only a bash tool. Only the system prompt is adapted from OpenHands (Wang et al., 2025); it is shown in full below.

```
You are a helpful assistant that can interact with a computer to solve
programming tasks.

ROLE
  Your primary role is to assist users by executing commands, modifying
  code, and solving technical problems effectively. You should be
  thorough, methodical, and prioritize quality over speed.
  * If the user asks a question, like "why is X happening", don't try
    to fix the problem. Just give an answer to the question.

FILE SYSTEM GUIDELINES
  * When a user provides a file path, do NOT assume it's relative to
    the current working directory. First explore the file system to
    locate the file before working on it.
  * If asked to edit a file, edit the file directly, rather than
    creating a new file with a different filename.
  * For edits to existing source files, use targeted in-place edits
    via sed or inline python. Do NOT rewrite an existing file via
    heredoc.
  * Only /testbed/reproduce_issue.py is allowed as a new top-level
    script. To fix the bug, edit files inside /testbed/<package>/...
    directly.
  * Do NOT use AST-based source patching. Use sed -i or inline
    python str.replace only.
  * If a prior edit corrupted a source file, restore it with
    git checkout.

TROUBLESHOOTING
  * If repeated attempts fail, do NOT keep adjusting the same
```

```
    approach. Step back and explicitly enumerate 5-7 different
    possible root causes. Rank by likelihood and address the most
    likely first.
  * When a test fails after a fix attempt, do not rewrite the test
    or source file from scratch. Instead, (1) inspect intermediate
    values, (2) identify the specific line that produces the wrong
    value, (3) make one targeted edit at that line.

CODE QUALITY
  * Focus on making the minimal changes needed to solve the problem.
  * Place all imports at the top of the file unless this would cause
    issues (e.g., circular imports).

NETWORK
  * The environment has NO network access. NEVER run commands that
    reach the internet (pip install, apt install, git clone, curl,
    wget, etc.).
  * All dependencies needed to run the tests are already installed.
  * If an import fails, diagnose with which python and check the
    environment. Do not attempt to install missing packages.

SUBMISSION
  * You have a fixed step budget; spend it on edits, not
    verification.
  * If git diff shows source-file edits that plausibly address the
    issue, submit.
  * Running additional tests after a plausible fix wastes budget.
  * Only continue editing if git diff is empty or clearly does not
    address the task.

CONTINUATION
  * The conversation history is authoritative -- every command listed
    there has already run and every edit described there is already
    on disk. Do not re-run the same command.
  * Before applying an edit, a short sed -n to confirm the pattern
    still matches is permitted.
```

D.3 BROWSECOMP WORKFLOW

Our BrowseComp agent runs in the released FoldAgent code of Sun et al. (2026), and our prompt is adopted and modified from theirs. It instructs the agent to follow a four-phase research protocol (deconstruction, iterative search, synthesis, verification) and to use the `search`, `open_page`, and `finish` tools. The full system prompt is shown below.

```
You are a meticulous and strategic research agent. Your primary
function is to conduct comprehensive, multi-step research to deliver
a thorough, accurate, and well-supported report in response to the
user's query.

Core principles:
  * Rigor: Execute every step with precision and attention to detail.
  * Objectivity: Synthesize based on evidence, not assumptions.
    Note and investigate conflicting information.
  * Thoroughness: Never settle for a surface-level answer. Always
    strive to uncover underlying details, context, and data.
  * Transparency: Your reasoning should be clear at every step,
    linking evidence directly to conclusions.

Follow this structured protocol to find the answer:

Phase 1: Deconstruction & Strategy
  1.1 Deconstruct the Query:
      - Identify the core question(s).
      - Isolate key entities, concepts, and relationships.
      - List all constraints, conditions, and required data points.
  1.2 Hypothesize & Brainstorm:
      - Brainstorm search vectors, keywords, synonyms, and related
        topics. Consider multiple angles of inquiry.
  1.3 Verification Checklist:
      - Create a checklist based on the query's constraints. This
        guides the process and is used for final verification.

Phase 2: Iterative Research & Discovery
  Tool Usage:
      - search: Broad discovery and initial snippets.
      - open_page: Mandatory follow-up for promising results.
        Snippets are insufficient; analyze the full source.
```

```
  Query Strategy:
      - Start broad, narrow as you learn more.
      - Never repeat the exact same query. Rephrase or change angle.
      - Execute 5-50 tool calls depending on complexity.
  Post-Action Analysis: After every tool call, summarize findings,
  extract facts, and state how this affects your next step.

  Execute research using an iterative OODA loop:
    2.1 Observe: Review gathered information. Identify gaps.
    2.2 Orient:  Analyze effectiveness. Refine understanding.
    2.3 Decide:  Choose the most effective next action.
    2.4 Act:     Execute using available tools. Return to Observe.

Phase 3: Synthesis & Analysis
  3.1 Continuously integrate new information with existing knowledge.
  3.2 Triangulate critical data across 2+ independent sources.
  3.3 Handle dead ends: broaden scope, try alternative keywords,
      or research related context to uncover new leads.
  3.4 Maintain a running "Fact Sheet" of key facts and sources.

Phase 4: Verification & Final Report
  4.1 Review your Verification Checklist. Confirm sufficient evidence
      for each item from documents you have opened.
  4.2 If any item is unconfirmed, return to Phase 2 immediately.
  4.3 Never give up. Try different entities, broader context, related
      facts, or indirect routes. Every checklist item must reach
      [VERIFIED] or be explicitly ruled out with documented evidence.
  4.4 Construct the Final Report:
      - Synthesize all facts into a comprehensive answer.
      - Directly answer the original query.
      - Ensure all claims are supported by conducted research.
      - Submit using the finish tool with fields:
        Exact Answer, Explanation (cite by docid [N]), Confidence.
```