Git Context Controller: Manage the Context of Agents by Agentic Git

Junde Wu[1] **Minhao Hu**[1] **Jiayuan Zhu**[1] **Shengda Zhu**[2] **Jiazhen Pan**[3]
Fenglin Liu[1] **Yuyuan Liu**[1] **Min Xu**[4,5] **Yueming Jin**[6,†]

[1]University of Oxford [2]University of Edinburgh [3]Stanford University
[4]Carnegie Mellon University [5]MBZUAI [6]National University of Singapore
[†]Corresponding author: ymjin@nus.edu.sg

Abstract

Large language model (LLM) agents have demonstrated strong capabilities in long-horizon tasks by interleaving reasoning with tool use. However, as these agents scale to complex workflows such as software engineering and open-ended research, context management becomes a fundamental bottleneck: interaction histories grow unbounded, become costly to maintain, and are difficult to reuse across sessions and agents. We introduce **Git-Context-Controller (GCC)**, a structured context management framework inspired by software version control systems. GCC elevates agent context from a transient token stream to a persistent, navigable memory workspace with explicit operations—`COMMIT`, `BRANCH`, `MERGE`, and `CONTEXT`, that enable milestone-based checkpointing, isolated exploration of alternative reasoning paths, and hierarchical retrieval of historical context. By organizing agent memory as a versioned file system, GCC allows agents to manage long-term goals, recover and transfer reasoning across sessions, and coordinate multi-trajectory problem solving in a principled manner. Empirically, agents equipped with GCC achieve state-of-the-art performance on both SWE-Bench and BrowseComp benchmarks. On SWE-Bench Verified, GCC improves task resolution by over 13% relative to strong long-context baselines and outperforms 26 existing open and commercial systems, reaching over 80% success rate. Code is available at `https://github.com/ImprintLab/git-context-controller`.

1 Introduction

LLM-based agents have been capable of interleaving internal chain-of-thought reasoning with external tool calls Wu et al. [2025]. Such architecture has shown strong performance in decision-making tasks, web interaction, and question answering benchmarks, providing a foundation for more sophisticated agents. In software engineering domains, frameworks like SWE-Agent Yang et al. [2024] used similar paradigm by integrating code generation, execution, and test loops to implement iterative software development (e.g., writing, compiling, debugging). Following this idea, production-grade tools such as Anthropic's Claude Code and Google's Gemini CLI bring LLM-based agents to the command line, enabling code completion, debugging, and search within a single session.

However, as LLM agents are increasingly deployed for long-horizon reasoning in complex, large-scale workflows, context management emerges as a fundamental bottleneck. A common issue observed in coding agents is that sessions gradually forget previous context and become increasingly costly as the context grows longer. Starting a new session typically erases the agent's memory of prior goals, user preferences, and task-specific instructions. As a result, users are forced to repeatedly provide the same context in every new session. Current solutions rely on several common strategies. A straightforward approach is to truncate older context once the token limit is reached. This risks discarding important

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

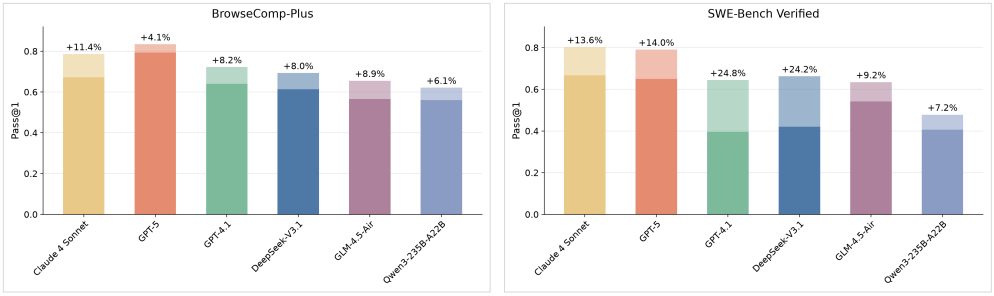

Figure 1: Results on BrowseComp-Plus and SWE-Bench Verified comparing baseline model performance with improvements achieved by equipping models with GCC. Baselines are the ReAct agent on BrowseComp-Plus and SWE-Agent on SWE-Bench Verified (Table 1); labels give the absolute gain in Pass@1.

historical details—especially problematic when the agent needs to revisit earlier decisions or maintain consistency across multi-step plans. A more balanced approach compresses earlier reasoning into high-level summaries or to-do lists, as seen in systems such as Claude Code and Gemini CLI. These systems use summary-based anchors for future reasoning and persist abstracted task state (e.g., via `agent.md` file). However, relying on simple compression removes fine-grained details and weakens the agent's ability to ground its actions in specific prior reasoning. A common and intuitive observation is that agents become "dumber" each time their context is compressed. In conclusion, context is currently either *too verbose* to be reusable or *too abstract* to support concrete continuation and extension.

These limitations highlight the need for a more principled and structured approach to how AI agents log, manage, and retrieve context. Our key insight is that the challenges faced by long-horizon agents closely mirror those encountered by software engineers managing complex, evolving codebases. Inspired by the success of Git in software version control, we propose *Git-Context-Controller (GCC)*, an agentic context control mechanism that elevates context management to an explicit abstraction layer. It organizes contextual information as a structured, version-controlled file system, and introduces a set of specialized commands designed to support logging, managing, and retrieving context across agentic workflows.

We implement this design through a standalone `Git-Context-Controller`, which structures agent context as a version-controlled file system under a unified `.GCC/` directory. Each project maintains a global roadmap (`main.md`), while each branch contains its own commit summaries, execution traces, and structured metadata. Agents interact with this controller through a small set of core commands: `COMMIT` to checkpoint meaningful progress, `BRANCH` to explore alternate strategies, `MERGE` to synthesize divergent reasoning paths, and `CONTEXT` to retrieve historical information at varying resolutions. These commands and data structures are provided to the agent and autonomously invoked to support long-horizon reasoning.

Such a design provides several complementary benefits. It enables multi-level context retrieval, allowing agents to access information at different levels of abstraction, ranging from high-level project plans to fine-grained OTA (Observation–Thought–Action) traces. Agents can flexibly navigate across these layers, starting from a coarse summary and drilling down into detailed execution histories whenever necessary, which makes past reasoning easy to trace and reuse. At the same time, the branching mechanism offers isolated workspaces for exploration, where agents can freely test new ideas or iterate without interfering with the main reasoning trajectory. This preserves focus while still allowing side explorations to be revisited or merged back into the primary workflow. Moreover, the framework naturally supports cross-agent and cross-session continuity: a new agent does not need to be re-instructed from scratch, and even an agent running on a different LLM or machine can seamlessly resume from the exact state left by its predecessor. This design facilitates smooth distribution and handover of agent-generated code and reasoning artifacts, in a manner analogous to how human developers collaborate through Git repositories.

Empirically, agents equipped with GCC achieve state-of-the-art (SOTA) performance on SWE-Bench and BrowseComp, two of the most widely used benchmarks for long-horizon coding and web-browsing reasoning. As shown in Fig. 1, on SWE-Bench, simply integrating GCC boosts models such as GPT-4.1 and DeepSeek by over 24%. Claude-4-Sonnet equipped with GCC further improves

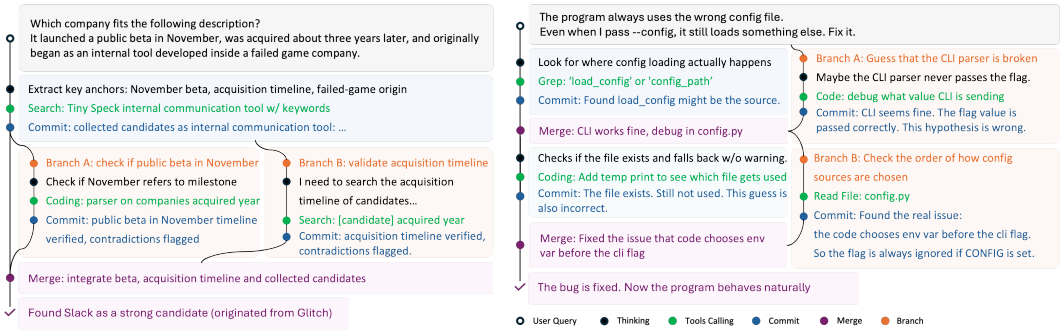

Figure 2: Illustration of GCC in action across two workflows: web-search reasoning and software debugging—showing how agents branch, explore, and merge structured context during long-horizon tasks.

by 13.6%, reaching 80.2%, which constitutes the current state-of-the-art result on SWE-Bench. On BrowseComp-Plus, GPT-5 with GCC achieves 83.4%, also establishing a new state-of-the-art performance on this benchmark.

In summary, our contributions are:

- We propose a novel view of agent memory as a dynamic, navigable codebase, complete with log files, branching histories, and metadata. This reframes context not just as passive history but as an evolving, queryable interface that supports both recall and structural reasoning.
- We introduce GCC, a structured context management framework for LLM agents that integrates version control semantics—such as `COMMIT`, `BRANCH`, and `MERGE` into the reasoning loop. GCC organizes agent memory into persistent, interpretable artifacts that support long-horizon workflows, architectural modularity, and reproducibility.
- When equipped with GCC, LLM-based agents achieve state-of-the-art empirical performance on both SWE-Bench and BrowseComp. In particular, on SWE-Bench, our method outperforms 26 existing systems (including both open-source and commercial models), achieving over 80% task resolution rate.

2 Method

We proposed `Git-Context-Controller` (GCC) as an abstraction layer for agent memory, consisting of a structured file system paired with a series of callable commands that agents use to externalize, organize, and retrieve their reasoning, as shown in Fig. 2. Inspired by version control systems like Git, GCC transforms the agent's ephemeral context into a persistent and navigable workspace. The core of GCC is a directory-based data structure that systematically organizes all historical context (GCC File System), together with a set of commands that enable agents to interact with and manipulate this structure (GCC Commands). Concretely, agents communicate with the controller via commands such as `COMMIT`, `BRANCH`, `MERGE`, and `CONTEXT`, which operate over a hierarchical workspace rooted at `.GCC/`. This workspace contains global planning artifacts (`main.md`), branch-specific execution traces (`log.md`), milestone-level summaries (`commit.md`), and structured metadata (`metadata.yaml`) that record architectural decisions and project states. In the following, we provide a detailed introduction to the structured file system and commands of GCC.

2.1 GCC File System

GCC organizes agent context into a structured directory rooted at `.GCC/`, reflecting a three-tiered hierarchy of reasoning: high-level planning, commit-level summaries, and fine-grained execution traces. Each file in this hierarchy plays a distinct role in tracking the agent's thoughts, progress, and architectural context. All files are plain-text and continuously updated through agent-invoked commands.

The overall layout of the file system is structured as following:

```
.GCC/
```

```
|-- main.md         # a global roadmap summarizing project
↪  high-level intent, milestones, and shared planning state across all branches
|-- branches/
   |-- <branch-name>/
      |-- commit.md    # recording the progress of each commit
      |-- log.md       # a detailed execution
↪  trace of OTA cycles, continuously recorded during the agent reasoning loop
      |-- metadata.yaml # a structured file storing branch-specific
↪  architectural and contextual metadata (file structures, dependencies, configs)
   |-- <another-branch>/...
```

In which, `main.md` sits at the root of the `.GCC/` directory and stores the global project roadmap. It records high-level project goals, key milestones, and the to-do list for development. This file is shared across all branches and serves as the canonical source of the project's overall intent. The agent is prompted to initialize this file with the project goal and initial to-do list at the beginning of the project. It may be revised later when a conclusion is reached, a major outcome is completed, or significant changes to the roadmap occur. Such updates are optionally triggered after `COMMIT`, `MERGE`, or `BRANCH` by the agents.

Each branch has its own directory under `branches/`, which contains three primary files. The first is `commit.md`, a structured summary log that captures the evolving progress of the branch. Each time the agent calls `COMMIT`, the controller appends a new entry to `commit.md` following a standardized template consisting of three blocks: (1) *Branch Purpose* – a reiteration of the overall project goal and the specific rationale for creating this branch (as defined at `BRANCH`); (2) *Previous Progress Summary* – a coarse-grained summary of the branch's history, generated by giving the last commit's Previous Progress Summary and This Commit's Contribution; and (3) *This Commit's Contribution* – a detailed narrative of what was achieved in the current commit.

The second file is `log.md`, which stores the fine-grained reasoning trace of the agent's execution. This includes every OTA (Observation–Thought–Action) cycle that occurs between commits. Each reasoning step is appended to `log.md` in real-time, forming a continuous trace of low-level decision-making. Upon committing, the relevant slice of this log is referenced to construct the summary in `commit.md`.

Finally, `metadata.yaml`, captures structured meta-level information. It includes details such as the current file structure of the project, per-file responsibilities, environment configurations, dependency graphs, or module interfaces. By default, commonly useful segments like `file_structure` and `env_config` are defined, while additional entries can be manually added by human users as needed. This file is updated on demand—typically during or after a `COMMIT`, when structural or configuration changes are detected. Together, these files provide a layered, interpretable view of agent reasoning from abstract goals to step-level execution.

2.2 GCC Commands

The `Git-Context-Controller` exposes a set of agent-callable commands that allow reasoning models to manage, structure, and retrieve context in a durable and inspectable way. These commands include `COMMIT`, `BRANCH`, `MERGE`, and `CONTEXT`. These commands' function and usage are given to the agents in the system prompts, then the agents are encouraged to use them when needed. For example, when the agent reflects on its reasoning and detects a shift in direction, it would evaluate whether a `BRANCH` is warranted. When a reasoning subgoal is achieved, it is encouraged to call `COMMIT` and summarize the step. Below, we detail the purpose, usage, and implementation of each command.

`COMMIT <summary>` The `COMMIT` command is called when the agent identifies that its recent reasoning has resulted in a coherent and meaningful milestone, such as implementing a function, completing a test, or resolving a subgoal. Once invoked, the controller performs a structured update across multiple files within the current branch directory.

Specifically, let the agent's recent execution trace be denoted as $\mathcal{H}_t$. The `COMMIT` operation transforms this transient reasoning history into a persistent memory record

$$\mathcal{M}_t = (I_t, S_t, D_t) = \texttt{COMMIT}(\mathcal{H}_t, S_{t-1}),$$

where I_t represents the branch intent, S_t is a regenerated coarse-grained summary combining the previous commit summary S_{t-1} with the newly completed work, and D_t is a detailed description of

the specific progress achieved since the last commit. When the global project plan evolves, `COMMIT` further induces an update of the global roadmap `main.md`. Finally, the memory update and code changes are consolidated into a versioned state using a Git commit with message S_t. Through this transformation, a loose sequence of observation–thought–action steps is converted into a coherent and retrievable memory unit that supports long-horizon progress tracking and rollback.

`BRANCH <name>` The `BRANCH` command is called when the agent detects a meaningful divergence in direction, such as exploring an alternative algorithm, implementing a parallel module, or testing a new design hypothesis.

Let the current memory state be represented by the latest committed record $\mathcal{M}_{t-1}$. When the command `BRANCH <name>` is issued, a new branch-specific execution state is created as: $\mathcal{B}_t^{(name)} = \texttt{BRANCH}(\mathcal{M}_{t-1})$, which initializes an empty observation–thought–action trace $\mathcal{H}_t^{(name)}$ stored in `log.md`, together with a new structured memory file `commit.md` that records the intent and motivation of the branch. This transformation establishes an isolated reasoning trajectory in which alternative hypotheses or experimental workflows can be explored independently from the mainline history, while remaining fully inspectable and reversible through the same commit-based memory mechanism.

`MERGE <branch>:` The `MERGE` command is used when a branch has reached a conclusion and its results are ready to be integrated into the main plan. Before merging, the controller automatically calls `CONTEXT` on the target branch to surface its historical summaries and planning rationale.

Let the current branch memory state be $\mathcal{M}_t = (I_t, S_t, D_t, \mathcal{H}_t)$ and the target branch memory state be $\mathcal{M}_t^{(b)} = (I_t^{(b)}, S_t^{(b)}, D_t^{(b)}, \mathcal{H}_t^{(b)})$. When the command `MERGE` is issued, a new unified memory state is produced as $\mathcal{M}_{t+1} = \texttt{MERGE}(\mathcal{M}_t, \mathcal{M}_t^{(b)})$, where the updated memory record stored in `commit.md` is defined by

$$(S_{t+1}, D_{t+1}) = \mathcal{F}_{\text{merge}}\big((S_t, D_t), (S_t^{(b)}, D_t^{(b)})\big).$$

Here, $\mathcal{F}_{\text{merge}}(\cdot)$ denotes a synthesis operator that integrates the branch purpose and progress from both branches into a unified summary S_{t+1} and a detailed description D_{t+1} explaining the rationale and outcome of the merge. The global planning state `main.md` is simultaneously updated as $\texttt{main.md}_{t+1} = \mathcal{G}(\texttt{main.md}_t, \mathcal{M}_t^{(b)})$, reflecting the impact of the merged branch on the overall roadmap and future milestones.

The execution traces stored in `log.md` are combined as $\mathcal{H}_{t+1} = \mathcal{H}_t \cup \mathcal{H}_t^{(b)}$, with explicit origin annotations to preserve the provenance of observation–thought–action steps. Finally, the unified memory state $\mathcal{M}_{t+1}$ is checkpointed through a new Git commit, ensuring that symbolic memory and executable artifacts are synchronized and versioned.

`CONTEXT <options>:` The `CONTEXT` command allows agents to retrieve memory at multiple levels of granularity, from global overviews to fine-grained token-level execution traces. This supports both reflective reasoning and task continuation across sessions. Agents are required to call `CONTEXT` in specific scenarios, such as when a new agent resumes an ongoing task, or before the MERGE command. Besides that, the agent is able to call `CONTEXT` proactively whenever it finds context retrieval necessary.

When the agent issues the `CONTEXT` command, the controller returns a structured snapshot of the current project state, analogous to a `git status` view over the memory directory. This snapshot exposes the global project purpose and milestone progress derived from `main.md`, together with the set of available branches.

For branch-level inspection, the agent may request `CONTEXT --branch <branch>`, which retrieves the branch intent and the latest progress summary stored in `commit.md`, along with a bounded window of recent commit records. Let the ordered commit history of a branch be denoted as $\{\mathcal{M}_i\}_{i=1}^{T}$. The returned view corresponds to a windowed projection $\mathcal{V}_k = \{\mathcal{M}_i\}_{i=k}^{k+K}$, where K is a fixed context budget and k is controlled by scrolling operations. This design ensures that long reasoning histories can be traversed incrementally without exceeding the agent's context capacity. More fine-grained retrieval is supported through specialized queries. The command `CONTEXT --commit <hash>` returns the complete structured memory record associated with a specific commit, while `CONTEXT --log` exposes a windowed segment of the execution trace $\mathcal{H}$ recorded in `log.md`

Model	Peak Length	Max #Token	BrowseComp-Plus Pass@1	BrowseComp-Plus Tool Calls	SWE-Bench Verified Pass@1	SWE-Bench Verified Tool Calls
ReAct Agent						
GPT-5	32K	32K	0.793	14.2	0.718	42.6
GPT-4.1	32K	32K	0.640	5.6	0.486	28.7
DeepSeek-V3.1	32K	32K	0.613	10.6	0.610	53.2
GLM-4.5-Air	32K	32K	0.566	11.1	0.576	51.2
Qwen3-235B-A22B	32K	32K	0.560	12.8	0.344	32.1
Claude 4 Sonnet	32K	32K	0.672	13.6	0.682	48.3
SWE Agent						
GPT-5	32K	32K	-	-	0.650	70.3
GPT-4.1	32K	32K	-	-	0.396	68.5
DeepSeek-V3.1	32K	32K	-	-	0.420	72.2
GLM-4.5-Air	32K	32K	-	-	0.542	71.7
Qwen3-235B-A22B	32K	32K	-	-	0.406	60.6
Claude 4 Sonnet	32K	32K	-	-	0.666	62.4
Summary Agent						
GPT-5	32K	32K × 100	0.765	16.4	0.690	52.0
GPT-4.1	32K	32K × 100	0.633	12.3	0.474	49.3
DeepSeek-V3.1	32K	32K × 100	0.592	18.6	0.626	55.6
GLM-4.5-Air	32K	32K × 100	0.565	14.3	0.566	51.0
Qwen3-235B-A22B	32K	32K × 100	0.578	10.4	0.378	44.6
Claude 4 Sonnet	32K	32K × 100	0.685	11.8	0.666	47.1
Folding Agent						
GPT-5	32K	32K × 100	0.815	20.1	0.746	95.3
GPT-4.1	32K	32K × 100	0.665	16.3	0.626	88.4
DeepSeek-V3.1	32K	32K × 100	0.640	18.2	0.616	96.6
GLM-4.5-Air	32K	32K × 100	0.595	19.7	0.596	92.9
Qwen3-235B-A22B	32K	32K × 100	0.585	20.3	0.366	82.7
Claude 4 Sonnet	32K	32K × 100	0.720	22.6	0.740	84.1
GCC Agent						
GPT-5	32K	32K × 100	**0.834** (+1.9)	24.5	**0.790** (+4.4)	101.5
GPT-4.1	32K	32K × 100	**0.722** (+5.7)	21.0	**0.644** (+1.8)	91.3
DeepSeek-V3.1	32K	32K × 100	**0.693** (+5.3)	19.2	**0.662** (+4.6)	93.6
GLM-4.5-Air	32K	32K × 100	**0.655** (+6.0)	26.6	**0.634** (+3.8)	95.1
Qwen3-235B-A22B	32K	32K × 100	**0.621** (+3.6)	21.4	**0.478** (+11.2)	89.3
Claude 4 Sonnet	32K	32K × 100	**0.786** (+6.6)	28.8	**0.802** (+6.2)	98.8

Table 1: **Performance on BrowseComp-Plus (N=150) and SWE-Bench Verified (N=500).** Green numbers give the absolute gain in Pass@1 (points) over the Folding Agent with the same backbone.

using the same sliding-window mechanism. System-level metadata, such as file structure and environment configuration, is accessed through `CONTEXT --metadata <segment>`, which retrieves the corresponding portion of `metadata.yaml`.

3 Experiment

3.1 Datasets

Deep Research: BrowseComp-Plus. For research-oriented tasks, we rely on BrowseComp-Plus (BC-Plus) [Chen et al., 2025], an extension of BrowseComp enriched with verified targets. High-quality supervision is particularly important in this domain, yet most existing web-research datasets are not publicly released [Qiao et al., 2025, Li et al., 2025]. We evaluate GCC on all instances of the dataset. The agent interacts with two tools, `search(query, topk)` and `open_page(url)`. All retrieval is performed with Qwen3-Embed-8B.

Agentic SWE: SWE-Bench and SWE-Benchlite. For software engineering, we follow the widely used SWE-Bench benchmark [Jimenez et al., 2024a], which tests an agent's ability to produce correct patches for real-world bugs described in natural-language issue reports. A task consists of the buggy project snapshot and the patch generation requirement. In addition, we also report results on SWE-Benchlite [Jimenez et al., 2024b], a curated and self-contained subset of 300 higher-quality problems that has become standard in recent evaluations.

3.2 Implementation

We set the LLM context window to 32,768 tokens for all comparison models. For the fold agent, summary agent, and GCC, we allow up to 100 active summarized archives/branches, yielding a theoretical maximum accessible context of 3,276,800 tokens. In all experiments, the commit retrieval window of the `CONTEXT` operation is fixed to $K = 1$, as determined by the ablation study in Appendix D. When evaluated on SWE-Bench, GCC is integrated on top of the standard SWE-Agent framework.

3.3 Baselines

We compare GCC against four standard long-context agent baselines widely adopted in prior work. Each baseline is instantiated with the same set of backend models as in Table 1: GPT-5, GPT-4.1, DeepSeek-V3.1, GLM-4.5-Air, Qwen3-235B-A22B, and Claude 4 Sonnet. All agents operate under identical tool APIs, context budgets, and evaluation settings. **ReAct Agent** [Yao et al., 2022]. Maintains full interaction history with a ReAct-style reasoning–acting loop. We evaluate under a fixed 32K token budget (Peak Length and Max#Token). **SWE-Agent** [Jimenez et al., 2023]. The default open-source reference system for SWE-Bench. Iteratively generates patches using execution, editing, and testing tool calls while keeping uncompressed full context. Following prior work, SWE-Agent is evaluated on SWE-Bench Verified only. **Summary Agent** [Yu et al., 2025]. Uses summary-based context compression: when the context approaches the 32K limit, it produces a high-level summary and replaces earlier history. This represents the standard "summarize-when-full" policy. **Folding Agent**. Implements the Folding paradigm Sun et al. [2025], periodically compressing earlier reasoning into structured summaries ("folds") while maintaining a shorter active window. This provides a middle ground between pure retention and pure summarization. All four baselines use the same underlying models, data, tools, and evaluation protocol as GCC.

4 Experimental Results

Table 1 presents a comprehensive comparison across four long-context agent baselines, ReAct, SWE-Agent, Summary Agent, and Folding Agent, together with our proposed GCC agent, evaluated under identical model backends and tool APIs. We also compare GCC with industry memory retrieval methods in Appendix A and analyze its efficiency in Appendix C. Here, we present only the main results.

GCC surpasses all controlled baselines. Across all model backbones, GCC achieves the best Pass@1 on both BrowseComp-Plus and SWE-Bench Verified. For instance, on BrowseComp-Plus with GPT-5, GCC reaches a Pass@1 of 0.834, outperforming the next-best Folding Agent (0.815). This trend persists across smaller models: with GPT-4.1, GCC improves Pass@1 from 0.665 (Folding) to 0.722; with DeepSeek-V3.1, from 0.640 to 0.693; and similarly strong gains appear for GLM-4.5-Air, Qwen-235B-A22B, and Claude 4 Sonnet. These results demonstrate that GCC consistently enhances the underlying model's capability, regardless of backend architecture.

GCC yields even larger gains on SWE-Bench Verified. SWE-Bench Verified presents significantly greater complexity, yet GCC achieves the strongest performance across all backbones. Using GPT-5, GCC attains a Pass@1 of 0.790, exceeding the Folding Agent (0.746) and outperforming the Summary and ReAct agents by even larger margins. Similar improvements appear across other models: for example, GCC improves Claude 4 Sonnet from 0.740 to 0.802, and DeepSeek-V3.1 from 0.616 to 0.662. These uniform improvements on a challenging benchmark confirm GCC's advantage in handling long-horizon software reasoning.

GCC allocates substantially more computation and interaction. A distinguishing characteristic of GCC is its increased tool-call frequency. On BrowseComp-Plus with GPT-5, GCC uses 24.5 tool calls, compared to 20.1 for the Folding Agent and just 14.2 for the long-context ReAct baseline. This trend becomes even more pronounced on SWE-Bench Verified, where GCC increases tool calls from 95.3 (Folding Agent) to 101.5. Similar jumps occur across all model backbones. These results indicate that GCC not only compresses and manages context better but also learns to allocate interaction capacity adaptively—probing deeper, exploring more paths, and conducting more thorough reasoning during thinking.

Tool	LLM	% Resolved	Avg. $ Cost	Avg. # Tokens	% Correct Location		
					Line	Function	File
CodeStory Aide [2024]	GPT-4o+ Claude 3.5 S	129 (43.00%)	$2.21	-	41.7%	58.7%	72.0%
Bytedance MarsCode Liu et al. [2024]	NA	118 (39.33%)	-	-	42.7%	58.0%	79.7%
Honeycomb [2024]	NA	115 (38.33%)	-	-	44.3%	57.0%	69.3%
MentatBot [2024]	GPT-4o	114 (38.00%)	$2.18	-	37.3%	53.3%	69.3%
Gru [2024]	NA	107 (35.67%)	-	-	38.3%	54.3%	75.0%
Isoform [2024]	NA	105 (35.00%)	-	41,963	38.7%	55.3%	72.0%
SuperCoder2.0 [2024]	NA	102 (34.00%)	-	-	41.7%	63.7%	65.7%
Alibaba Lingma Agent [2024]	GPT-4o+ Claude 3.5 S	99 (33.00%)	$4.22	-	40.0%	58.7%	75.0%
Factory Code Droid [2024]	NA	94 (31.33%)	-	-	36.7%	55.7%	72.7%
Amazon Q Developer-v2 [2024]	NA	89 (29.67%)	-	-	40.3%	52.0%	74.3%
SpecRover Ruan et al. [2024]	GPT-4o+ Claude 3.5 S	93 (31.00%)	$0.79	-	-	-	-
CodeR Chen et al. [2024]	GPT-4	85 (28.33%)	$3.34	323,802	35.7%	52.3%	67.0%
MASAI Arora et al. [2024]	NA	84 (28.00%)	-	-	38.7%	56.3%	75.0%
SIMA [2024]	GPT-4o	83 (27.67%)	$1.46	-	37.0%	54.0%	79.0%
IBM Research Agent-101 [2024]	NA	80 (26.67%)	-	-	39.7%	56.7%	73.3%
OpenCSG StarShip [2024]	GPT-4	71 (23.67%)	-	-	39.0%	61.7%	90.7%
Amazon Q Developer [2024]	NA	61 (20.33%)	-	-	34.0%	43.7%	71.7%
RepoUnderstander Ma et al. [2024]	GPT-4	64 (21.33%)	-	-	-	-	-
AutoCodeRover-v2 [2024]	GPT-4o	92 (30.67%)	-	-	35.0%	52.3%	69.3%
RepoGraph [2024]	GPT-4o	89 (29.67%)	-	-	36.7%	51.3%	71.0%
Moatless [2024]	Claude 3.5 S	80 (26.67%)	$1.75	-	38.7%	54.7%	78.7%
	GPT-4o	74 (24.67%)	$1.28	-	36.0%	52.0%	73.0%
OpenDevin+CodeAct v1.8 [2024]	Claude 3.5 S	80 (26.67%)	$1.14	-	38.0%	49.7%	67.3%
Aider Gauthier [2024]	GPT-4o+ Claude 3.5 S	79 (26.33%)	-	-	35.3%	50.0%	69.7%
SWE-agent Yang et al. [2024]	Claude 3.5 S	69 (23.00%)	$1.62	521,208	40.7%	54.3%	72.0%
	GPT-4o	55 (18.33%)	$2.53	498,346	29.3%	42.3%	58.3%
	GPT-4	54 (18.00%)	$2.51	245,008	30.7%	45.3%	61.0%
AppMap Navie [2024]	GPT-4o	65 (21.67%)	-	-	29.7%	44.7%	59.7%
AutoCodeRover Zhang et al. [2024]	GPT-4	57 (19.00%)	$0.45	38,663	29.0%	42.3%	62.3%
RAG Yang et al. [2024]	Claude 3 Opus	13 (4.33%)	$0.25	-	22.0%	30.0%	57.0%
	GPT-4	8 (2.67%)	$0.13	-	12.7%	23.3%	47.3%
	Claude-3 Opus	9 (3.00%)	-	-	16.7%	24.3%	46.7%
	GPT-3.5	1 (0.33%)	-	-	6.3%	11.3%	27.3%
AgentLess Xia et al. [2024]	GPT-4o	96 (32.00%)	$0.70	78,166	35.3%	52.0%	69.7%
GCC	GPT-3.5	**90 (30.00%)**	$0.57	386,490	35.7%	51.3%	69.3%
	GPT-4o	**138 (46.00%)**	$1.13	546,798	42.0%	59.3%	76.0%
	Claude 3.5 S	**144 (48.00%)**	$1.21	468,549	44.3%	61.7%	78.7%

Table 2: Results on SWEBench-Lite. The '–' symbol denotes missing / unreleased information needed to compute the corresponding value. Claude 3.5 S denotes Claude 3.5 Sonnet.

4.1 GCC Performance-Efficiency on Coding

We report an extended analysis of the SWE-Benchlite results in Table 2, comparing GCC against 26 state-of-the-art agentic coding systems spanning open-source methods, commercial closed-source agents, hybrid GPT–Claude ensembles, and retrieval-only baselines. Because these systems differ widely in model backend, memory strategy, and interaction protocol, SWE-Benchlite provides a stress test of *general-purpose coding competence* rather than controlled ablations (An apple-to-apple Performance-Efficiency comparison is shown in Appendix B.). Several fine-grained patterns emerge.

GCC establishes a new state of the art across all competing systems. GCC achieves a resolution rate of **48.00%**, the highest among all 26 systems. The next best performer, CodeStory Aide (43.00%), uses a hybrid GPT-4o + Claude 3.5 Sonnet backend but does not release intermediate reasoning or memory structure. Other top-tier commercial agents: ByteDance MarsCode (39.33%), Honeycomb (38.33%), and MentatBot (38.00%), all fall short by 5–10 percentage points. This margin is remarkable because many of these competing systems rely on fine-tuned internal components or private coding infrastructure, whereas GCC uses a transparent protocol layered onto an off-the-shelf LLM.

GCC closes the gap between proprietary agents and model-agnostic open systems. A notable trend in Table 2 is that several commercial systems (e.g., MarsCode, SIMA, Lingma, Honeycomb) cluster within a narrow accuracy band of 33–40%. Their performance gains primarily come from heavy engineering, test harness integration, and custom model routing: advantages unavailable to typical open-source agents. In contrast, GCC surpasses all of them while remaining purely protocol-driven. This suggests that structured, versioned memory confers benefits comparable to (or exceeding) architecture-level fine-tuning and proprietary engineering.

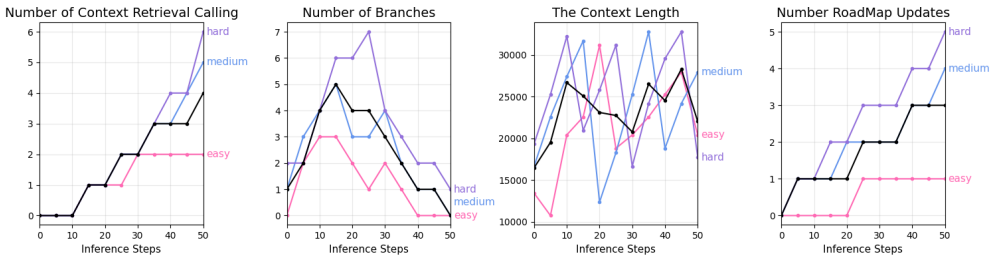

Figure 3: Number of context retrieval calling, number of branches, the context length and number of RoadMap updates with the increase of the inference steps

GCC achieves high performance without excessive inference cost. Despite the increased structure and deeper exploration that GCC encourages, the cost per task remains modest. For example, on Claude 3.5 Sonnet, the average cost is $1.21—comparable to, and often lower than, many top-performing agents such as SWE-Agent (up to $2.53) or CodeR (up to $3.34). This shows that the performance gain is not merely due to brute-force trial-and-error or excessive tool usage; rather, GCC induces structured tool usage that is cost-efficient relative to the quality of patches produced.

4.2 Ablation and Analysis

We perform an ablation study to evaluate the contribution of each component in GCC, including the RoadMap (main.md), Detailed Logs (log.md), Meta Data (metadata.yaml), CONTEXT retrieval, and BRANCH & MERGE, and COMMIT operations. Results on SWE-Bench Verified are reported in Table 3. A more detailed ablation is provided in Appendix D.

Starting from a raw Claude-4-Sonnet without structured memory (67.2%), adding RoadMap and COMMIT improves performance to 69.1%, showing that milestone-based checkpointing alone provides limited gains. Introducing Detailed Logs and CONTEXT retrieval yields a substantial improvement to 75.3%, highlighting the importance of preserving fine-grained reasoning traces and enabling explicit historical access. Further incorporating Meta Data increases performance to 77.8%, suggesting that structured architectural context supports consistency in long-horizon reasoning. The full system with BRANCH and MERGE achieves the best result of 80.2%, demonstrating that isolated exploration and controlled synthesis of alternative reasoning paths are critical for complex tasks.

We further analyze the evolution of internal behaviors during inference in Figure 3, including the number of CONTEXT calls, active branches, context length, and RoadMap updates. To enable controlled comparison across levels of challenge, we group evaluation tasks into *easy*, *medium*, and *hard*. For SWE datasets, difficulty is derived from the human time-to-resolution metadata originally provided with SWE-Bench: issues historically fixed within 15 minutes are *easy* (194 tasks), those requiring 15–60 minutes are *medium* (261 tasks), and those taking more than an hour are *hard* (45 tasks).

We find in Figure 3, CONTEXT retrieval increases with inference steps and task difficulty, indicating adaptive reliance on historical memory. The number of branches rises in intermediate stages and decreases as the agent converges, reflecting exploration followed by consolidation. Context length shows non-monotonic variation due to periodic summarization and retrieval, preventing unbounded growth. RoadMap updates increase steadily, especially for harder tasks, revealing continuous refinement of the global plan. These results show that each GCC component contributes incrementally, while the complete system induces structured behaviors of exploration, reflection, and consolidation, leading to more effective long-horizon reasoning.

RoadMap	Detailed Logs	Meta Data	CONTEXT	BRANCH & MERGE	COMMIT	**SWE** Verified
						67.2
✓					✓	69.1
✓	✓		✓		✓	75.3
✓	✓	✓	✓		✓	77.8
✓	✓	✓	✓	✓	✓	80.2

Table 3: Ablation study on GCC File System and Commands.

5 Conclusion

We introduced GCC, a structured context management framework that organizes agent memory using version control-inspired operations. GCC enables agents to persist, retrieve, and explore reasoning trajectories through committing, branching, and merging. Experiments show that GCC yields consistent performance gains and achieves SOTA results, suggesting that structured, Git-style memory is a key ingredient for effective long-horizon reasoning in autonomous agents. Limitations and broader impacts are discussed in Appendix G.

Acknowledgments and Disclosure of Funding

Funding. This work is supported in part by the MBZUAI-WIS Joint Program for Artificial Intelligence Research and by the Ministry of Education Tier 2 grant, Singapore (MOE-000935-01).

Competing interests. The authors declare no competing interests.

References

Junde Wu, Jiayuan Zhu, Yuyuan Liu, Min Xu, and Yueming Jin. Agentic reasoning: A streamlined framework for enhancing llm reasoning with agentic tools. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 28489–28503, 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. *arXiv preprint arXiv:2405.15793*, 2024.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent. *ArXiv*, abs/2508.06600, 2025.

Zile Qiao, Guoxin Chen, Xuanzhong Chen, Donglei Yu, Wenbiao Yin, Xinyu Wang, Zhen Zhang, Baixuan Li, Huifeng Yin, Kuan Li, Rui Min, Minpeng Liao, Yong Jiang, Pengjun Xie, Fei Huang, and Jingren Zhou. Webresearcher: Unleashing unbounded reasoning capability in long-horizon agents. In *arXiv preprint*, 2025.

Kuan Li, Zhongwang Zhang, Huifeng Yin, Rui Ye, Yida Zhao, Liwen Zhang, Litu Ou, Dingchu Zhang, Xixi Wu, Jialong Wu, Xinyu Wang, Zile Qiao, Zhen Zhang, Yong Jiang, Pengjun Xie, Fei Huang, and Jingren Zhou. Websailor-v2: Bridging the chasm to proprietary agents via synthetic data and scalable reinforcement learning. In *arXiv preprint*, 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*, 2024a. URL https://openreview.net/forum?id=VTF8yNQM66.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench Lite. https://www.swebench.com/lite.html, 2024b.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *ArXiv*, abs/2210.03629, 2022.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *ArXiv*, abs/2310.06770, 2023.

Hongli Yu, Tinghong Chen, Jiangtao Feng, Jiangjie Chen, Weinan Dai, Qiying Yu, Ya-Qin Zhang, Wei-Ying Ma, Jingjing Liu, Mingxuan Wang, and Hao Zhou. Memagent: Reshaping long-context llm with multi-conv rl-based memory agent. *ArXiv*, abs/2507.02259, 2025.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon llm agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025.

CodeStory. Aide by Codestory. https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240702_codestory_aide_mixed, 2024.

Yizhou Liu, Pengfei Gao, Xinchen Wang, Chao Peng, and Zhao Zhang. Marscode agent: Ai-native automated bug fixing. *arXiv preprint arXiv:2409.00899*, 2024.

Honeycomb. Honeycomb. https://honeycomb.sh, 2024.

Mentat. MentatBot: New SOTA Coding Agent, Available Now. https://mentat.ai/blog/mentatbot-sota-coding-agent, 2024.

Gru. The Road to Ultimate Pull Request Machine. https://gru.ai/blog/road-to-ultimate-pull-request-machine/, 2024.

Isoform. Isoform. https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240829_Isoform, 2024.

SuperAGI. SuperCoder. https://superagi.com/supercoder/, 2024.

Alibaba Lingma. Lingma Agent. https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240622_Lingma_Agent, 2024.

Factory. Factory Bringing Autonomy to Software Engineering. https://www.factory.ai/, 2024.

Amazon. Amazon Q Developer The most capable generative AI–powered assistant for software development. https://aws.amazon.com/q/developer//, 2024.

Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. Specrover: Code intent extraction via llms. *arXiv preprint arXiv:2408.02232*, 2024.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. Coder: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*, 2024.

Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. Masai: Modular architecture for software-engineering ai agents. *arXiv preprint arXiv:2406.11638*, 2024.

SIMA. Alex SIMA. https://github.com/swe-bench/experiments/tree/main/evaluation/lite/20240706_sima_gpt4o, 2024.

IBM Research. Agent-101: A Software Engineering Agent for Code Assistance developed by IBM Research. https://github.com/swe-bench/experiments/blob/main/evaluation/lite/20240612_IBM_Research_Agent101/README.md/, 2024.

OpenCSG. OpenCSG StarShip. https://opencsg.com/product?class=StarShip/, 2024.

Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. How to understand whole software repository? *arXiv preprint arXiv:2406.01422*, 2024.

AutoCodeRover. AutoCodeRover Autonomous Software Engineering. https://autocoderover.dev/, 2024.

RepoGraph. RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph. https://github.com/ozyyshr/RepoGraph, 2024.

Moatless. Moatless Tools. https://github.com/aorwall/moatless-tools, 2024.

OpenDevin. OpenDevin: Code Less, Make More. https://github.com/OpenDevin/OpenDevin/, 2024.

Paul Gauthier. Aider is ai pair programming in your terminal. https://aider.chat/, 2024.

AppMap. AppMap speedruns to the top of the SWE Bench Leaderboard. https://appmap.io/blog/2024/06/20/appmap-navie-swe-bench-leader/, 2024.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement, 2024.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Supermemory Team. Supermemory. https://github.com/supermemoryai/supermemory, 2024. Official website: https://supermemory.ai.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

Volcengine. Openviking. https://github.com/volcengine/OpenViking, 2024.

Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

LangChain. Langchain. https://github.com/langchain-ai/langchain, 2024.

Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of llm agents. *arXiv preprint arXiv:2402.17753*, 2024.

Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory. *arXiv preprint arXiv:2410.10813*, 2024. URL https://arxiv.org/abs/2410.10813.

Zihao Cheng, Weixin Wang, Yu Zhao, Ziyang Ren, Jiaxuan Chen, Ruiyang Xu, Shuai Huang, Yang Chen, Guowei Li, Mengshi Wang, Yi Xie, Ren Zhu, Zeren Jiang, Keda Lu, Yihong Li, Xiaoliang Wang, Liwei Liu, and Cam-Tu Nguyen. Lifebench: A benchmark for long-horizon multi-source memory. *arXiv preprint arXiv:2603.03781*, 2026. URL https://arxiv.org/abs/2603.03781.

A Comparison with Industry Memory Management Methods

A clear difference between GCC and currently industry-used memory management methods is that GCC is designed around a memory interface that is already native to LLMs, which is the Git-style operations. As a result, the model can use these operations actively and fluently during reasoning, instead of treating memory as a separate retrieval backend.

We conduct an experiment to support this distinction clearly. In Table 4, we compare GCC against 5 representative memory management methods (Supermemory Supermemory Team [2024], Mem0 Chhikara et al. [2025], OpenViking Volcengine [2024], Letta Packer et al. [2023], and LangMem LangChain [2024]) across three agent-memory benchmarks (Locomo Maharana et al. [2024], LongMemEval Wu et al. [2024], LifeBench Cheng et al. [2026]). We evaluate retrieval accuracy, context processing time, recall latency, and total context tokens. Across these benchmarks, GCC consistently achieves the best overall accuracy-efficiency trade-off. Methods such as Supermemory and OpenViking rely on graph-based or vectorized retrieval pipelines, which incur substantially higher context processing time. Letta adopts a hierarchical memory design, but its structure is still more complex and less aligned with native LLM operations, leading to additional overhead during context processing.

In contrast, GCC remains competitive with lightweight methods such as LangMem in terms of processing efficiency, while achieving significantly higher retrieval accuracy. Moreover, GCC consistently uses among the fewest context tokens, enabled by a design that reveals only concise summaries and retrieves detailed information on demand via Git-style operations.

This demonstrates that GCC achieves a higher signal-to-noise ratio in memory compression, allowing the agent to reliably locate precise information from a minimal exposed context, rather than relying on larger but noisier memory representations.

B Apples-to-apples performance-efficiency comparison under the same LLM models

We provide an apples-to-apples comparison in Table 5. Specifically, we evaluate eight strong and widely used agent frameworks under controlled settings, where each method is paired with the same LLM backbones (GPT-3.5, GPT-4o, and Claude 3.5 Sonnet), and compare them on SWE-Bench Lite. We report resolved cases, total budget consumed, and Resolved / Cost to measure efficiency in token usage.

From the results, GCC achieves the highest resolved rate while also delivering the best Resolved / Cost, under controlled settings where all methods share the same backbone models and comparable protocols. This shows that the gains come from the framework design itself, rather than differences in model capability or evaluation conditions

C Efficiency of GCC

Cost–performance trade-off is explicitly favorable for GCC To evaluate the GCC's cost–performance trade-off, we provide an extra cost-aware analysis: in Table 5, we report a cost-normalized metric (resolved / cost) on SWE-Bench Lite across 8 competitive agent frameworks. This metric captures how much improvement each method achieves per unit of compute. Our results show that GCC achieves state-of-the-art performance on this efficiency metric, indicating that the

Table 4: Comparison with industry memory management methods across long-horizon agent-memory benchmarks. Best results are in **bold**.

	LoCoMo				LongMemEval				LifeBench			
Method	Acc	CTX Time	Recall	Tokens	Acc	CTX Time	Recall	Tokens	Acc	CTX Time	Recall	Tokens
Supermemory	87.5	9677	1142	23,680	83.0	19373	1672	28,475	71.8	31454	870	38,545
Mem0	68.3	18493	516	25,902	69.5	35228	**462**	22,784	55.7	40779	679	45,213
OpenViking	85.6	39677	**220**	20,987	77.1	74897	685	18,543	67.8	150231	**642**	30,326
Letta (MemGPT)	75.1	1750	865	31,445	73.1	2863	1864	21,667	61.0	4863	1450	43,450
LangMem	61.6	**1018**	975	19,230	67.6	2279	1453	16,462	53.4	3268	977	28,092
GCC (Ours)	**88.1**	1326	643	**17,474**	**85.2**	**2136**	1355	**13,635**	**73.1**	**3184**	722	**20,417**

Table 5: Apples-to-apples comparison across agent frameworks under controlled backbones. Best results are in **bold**.

Framework	Backbone	Resolved	Avg. Cost ($)	Resolved / Cost
CodeStory Aide	GPT-3.5	82 (27.33%)	1.01	81.19
	GPT-4o	120 (40.00%)	2.05	58.54
	Claude-3.5 S	131 (43.67%)	2.37	55.27
OpenDevin	GPT-3.5	42 (14.00%)	0.59	71.19
	GPT-4o	76 (25.33%)	1.08	70.37
	Claude-3.5 S	80 (26.67%)	1.14	70.18
Bytedance MarsCode	GPT-3.5	75 (25.00%)	1.75	42.85
	GPT-4o	104 (34.66%)	3.02	34.43
	Claude-3.5 S	118 (39.33%)	3.46	34.10
MentaBot	GPT-3.5	79 (26.33%)	1.02	77.45
	GPT-4o	114 (38.00%)	2.18	52.29
	Claude-3.5 S	128 (42.67%)	2.56	50.00
CodeR	GPT-3.5	76 (25.33%)	2.87	26.48
	GPT-4o	99 (33.00%)	4.08	24.26
	Claude-3.5 S	105 (35.00%)	4.98	21.08
Alibaba Lingma Agent	GPT-3.5	62 (20.67%)	0.71	87.32
	GPT-4o	92 (30.67%)	1.27	72.44
	Claude-3.5 S	101 (33.67%)	1.40	72.14
SpecRover	GPT-3.5	48 (16.00%)	0.32	150.00
	GPT-4o	76 (25.33%)	0.63	120.63
	Claude-3.5 S	88 (29.33%)	0.81	108.64
SWE-agent	GPT-3.5	54 (18.00%)	2.51	21.51
	GPT-4o	55 (18.33%)	2.53	21.74
	Claude-3.5 S	69 (23.00%)	1.62	42.59
GCC (Ours)	GPT-3.5	90 (30.00%)	0.57	**157.89**
	GPT-4o	138 (46.00%)	1.13	**122.12**
	Claude-3.5 S	144 (48.00%)	1.21	**119.01**

additional tool calls are not wasteful overhead but economically justified reasoning steps. In other words, GCC does not merely spend more compute, it converts compute into successful resolutions more effectively than baselines.

In comparing with other competitive memory mechanism, GCC remains the most efficient In the table 4, we can see GCC consistently achieves the strongest accuracy while maintaining competitive latency and, importantly, the lowest context footprint. Notably, alternative memory structures that incur higher latency and produce larger context still underperform GCC in retrieval accuracy, indicating that the gains are not a byproduct of increased compute or context volume.

This efficiency stems directly from the design of GCC itself. COMMIT operations compress long interaction histories into structured summaries, effectively reducing context window pressure. In contrast to carefully designed but heavier retrieval pipelines (e.g., graph-based Supermemory or vector-RAG systems such as OpenViking), GCC leverages Git-style operations that are already well-internalized during training and are naturally usable by LLM-based agents. As a result, agents can invoke these primitives in a more efficient and fluent manner, leading to more effective memory usage.

Together, these results demonstrate that GCC's improvements arise from more effective and better-structured memory design, rather than increased computational budget.

Table 6: Crossed ablation over GCC components. Best performance is in **bold**.

RoadMap	Logs	Meta	Context	Branch & Merge	Commit	SWE-Verified
						67.2
✓					✓	69.1
✓	✓		✓		✓	75.3
✓	✓	✓	✓		✓	77.8
✓	✓	✓		✓	✓	76.2
✓			✓		✓	71.7
✓				✓	✓	70.9
✓			✓	✓	✓	75.3
✓	✓			✓	✓	74.6
✓	✓		✓	✓	✓	76.2
✓		✓		✓	✓	72.3
✓		✓	✓		✓	73.5
✓		✓	✓	✓	✓	74.1
✓	✓	✓	✓	✓	✓	**80.2**

Table 7: Ablation on CONTEXT retrieval window size K. Best results are in **bold**.

K	Resolved	Avg. Scroll Backward Freq.	Budget ($)	Resolved / Cost
$K=1$	**144 (48.00%)**	2.28	**1.21**	**119.01**
$K=2$	142 (47.33%)	2.12	1.38	102.90
$K=3$	144 (48.00%)	2.04	1.47	97.96
$K=4$	140 (46.67%)	1.68	1.60	87.50
$K=5$	142 (47.33%)	1.20	1.78	79.78

D Detailed Ablation

D.1 Crossed ablation over GCC components.

We provide a more detailed ablation study in Table 6, where we evaluate partially crossed combinations of all components. From the table, we observe that while individual modules provide incremental gains, their effects are complementary rather than independent, for instance, BRANCH & MERGE or CONTEXT alone have limited impact but become more effective when combined with structured logs. The full configuration achieves the best performance (80.2), indicating strong synergy.

D.2 Ablation over the retrieval window size K

We provide in Table 7 an ablation over the retrieval window size K, reporting resolved rate, average backward scrolling frequency, cost, and resolved-per-cost efficiency. Empirically, we observe that retrieving only the most recent commit (K=1) provides the best trade-off between relevance and efficiency. We can see when K is small but greater than 1 (e.g., K<3), including additional recent commits does not significantly reduce backward scrolling. This is because, in practice, when the agent scrolls, it is typically searching for distant historical context rather than slightly older adjacent commits. As a result, expanding K in this regime provides limited benefit.

When K becomes larger, although the frequency of explicit scrolling decreases, performance does not improve. Instead, retrieving multiple commits at once introduces a substantial amount of irrelevant or weakly relevant context, which contaminates the context window and reduces reasoning efficiency. This also increases token consumption per CONTEXT call, leading to higher computational cost without corresponding gains.

Based on these empirical findings, we set K=1 in all experiments, as it provides the best balance between performance, efficiency, and context quality.

E Analysis of GCC context retrieval behavior

Note that GCC does not rely solely on the CONTEXT command for retrieval. CONTEXT is designed as a structured exposure interface that surfaces organized memory, rather than a full retrieval mechanism by itself. The actual retrieval and search behavior is largely agent-native, leveraging standard tools such as keyword-based search (e.g., grep) over the exposed memory.

In practice, when the agent needs specific information, it does not sequentially scan multiple CONTEXT windows. Instead, it typically performs targeted search operations over the retrieved snapshots, for example:

```
CONTEXT --commit <hash> | grep -n -C 3 "keyword"
```

In this case, only the matched snippets are surfaced to the agent, rather than requiring full-window traversal. This behavior is not explicitly hard-coded, but emerges naturally because such operations are already well-internalized by LLM agents through pretraining and tool usage patterns. The solo CONTEXT command is used only when the agent does not yet know what to search for (i.e., exploratory access).

We believe this is a key reason why GCC achieves both strong performance and efficiency. GCC only focuses on organizing and exposing structured memory, while leaving retrieval and search to the agent's native capabilities. This results in a streamlined design with lower complexity and greater flexibility compared to systems that rely on explicitly engineered retrieval modules (e.g., RAG or semantic search pipelines), which often introduce additional overhead and rigid access patterns.

We further provide an analysis of retrieval behavior on SWE-Bench to better understand how agents actually access memory under GCC in Table 8. Specifically, we break down the average frequency of different retrieval patterns, including direct CONTEXT usage and CONTEXT followed by native keyword-based search (e.g., grep), across roadmap-, branch-, commit-, log-, and metadata-level views.

We observe that GCC is used primarily through a structured-view-plus-native-search pattern, rather than through raw sequential reading. In particular, CONTEXT --commit & grep, CONTEXT --log & grep, and CONTEXT --meta & grep are substantially more frequent than their raw counterparts, while direct CONTEXT --log access is rare. This reveals several important properties of GCC's retrieval behavior. First, the agent does not treat memory as a long text buffer to be scanned window by window. Instead, it first uses GCC's hierarchy to localize the likely region of relevance, for example, by identifying the appropriate roadmap state, branch, commit, or metadata segment, and only then performs targeted keyword search within that bounded snapshot. Second, the particularly high frequency of CONTEXT --commit & grep suggests that the commit is the dominant practical retrieval unit: it is detailed enough to preserve concrete reasoning and implementation progress, while still much more compact and navigable than full logs. Third, the large gap between CONTEXT --log & grep and raw CONTEXT --log indicates that log-level memory is usually accessed only after the agent already has a concrete hypothesis about what it is looking for, such as an error string, function name, config key, or file path. In other words, logs are not being traversed blindly; they are being queried selectively. Finally, the relatively frequent use of CONTEXT --meta and CONTEXT --meta & grep shows that a substantial portion of retrieval in SWE-style tasks is not about replaying old reasoning traces, but about recovering structured project state such as file layout, environment configuration, module responsibilities, and dependency information. Taken together, these results suggest that GCC improves efficiency not by introducing a heavier retrieval backend, but by exposing memory in a form that naturally supports the agent's existing search habits. GCC therefore delegates organization to the controller and fine-grained retrieval to the agent, which keeps the system simple, flexible, and efficient while still enabling strong long-horizon reasoning performance.

F Failure Case Analysis

We present a representative case in which GCC fails: a detail omitted during `COMMIT` becomes important only after new evidence appears.

The user reports:

```
On Windows, test_export_temp_file intermittently fails because
the generated temporary file cannot be deleted.
```

Table 8: Average frequency of GCC retrieval patterns.

Retrieval Pattern	Freq.	Retrieval Pattern	Freq.
`CONTEXT`	0.5	`CONTEXT & grep`	1.1
`CONTEXT --branch`	0.7	`CONTEXT --branch & grep`	1.6
`CONTEXT --commit`	0.9	`CONTEXT --commit & grep`	2.1
`CONTEXT --log`	0.2	`CONTEXT --log & grep`	1.7
`CONTEXT --meta`	1.7	`CONTEXT --meta & grep`	2.0

The agent runs the failing test and receives:

```
ResourceWarning: unclosed file <output.csv>
PermissionError: [WinError 32] The process cannot access output.csv
because it is being used by another process.
```

At this stage, the agent focuses on the final `PermissionError`. It reasons that Windows file deletion is more restrictive than Linux and treats the earlier `ResourceWarning` as an unrelated warning. It investigates the cleanup function and then commits:

```
Observed:
Temporary-file deletion fails on Windows with WinError 32.

Current hypothesis:
The cleanup routine does not handle Windows file locking correctly.

Next:
Inspect deletion order and consider retrying os.unlink().
```

The exact `ResourceWarning` and the function that opened `output.csv` remain in the raw execution log (`log.md`) but are omitted from the commit summary, because the agent does not yet consider them important.

Later, the agent obtains new evidence: the failure occurs only when `serialize_export()` raises an exception before completing. This should change the diagnosis, as the exception path may leave the file handle open, making the earlier `ResourceWarning` directly relevant. However, the agent retrieves the commit summary and sees only the Windows-locking hypothesis. It searches the stored context for `WinError`, `unlink`, and `cleanup`, but does not search for `ResourceWarning` or `unclosed file`, because those details are absent from the summary and therefore do not enter its current hypothesis. It consequently adds a retry loop around `os.unlink()`:

```
for _ in range(3):
    try:
        os.unlink(path)
        break
    except PermissionError:
        time.sleep(0.1)
```

The patch does not solve the problem, because the file handle remains open throughout all three retries. The correct fix would be to close the file in a `finally` block inside `serialize_export()`.

GCC fails here through a specific sequence:

1. The initial `COMMIT` treats `ResourceWarning` as irrelevant and omits it.
2. Later evidence makes the omitted warning diagnostic.
3. `CONTEXT` returns the earlier high-level conclusion rather than the overlooked detail.
4. The agent's search terms are derived from that conclusion, so it never locates the relevant raw-log entry.
5. The agent implements a patch for file deletion rather than file-handle management.

Keeping the complete test output in every commit would remove much of the benefit of compression, so the detail remains available only in the raw log. Recovery therefore depends on the agent recognizing what to search for; in this case, the information is stored but not found. A possible solution is to enable GCC to detect uncertainty in commit summaries and automatically preserve the supporting raw observations. We are developing this uncertainty-aware memory consolidation and retrieval mechanism and plan to release it in future work.

G Limitations and Broader Impacts

Limitations. GCC relies on the backbone LLM to decide when to invoke `COMMIT`, `BRANCH`, `MERGE`, and `CONTEXT`, and to write faithful commit summaries. Weaker models may under- or over-use these commands, and an inaccurate or incomplete summary can propagate to later steps or to agents that resume from it (see the failure case in Appendix F). Our evaluation covers software engineering (SWE-Bench Verified and Lite), deep research (BrowseComp-Plus), and agent-memory benchmarks (LoCoMo, LongMemEval, LifeBench); other settings, such as multimodal or embodied agents, or several agents writing to the same memory concurrently, are not evaluated. GCC increases the number of tool calls per task (Table 1), which adds latency even though cost-normalized efficiency remains favorable (Appendix C). We do not report error bars, because repeating every experiment across all backbones with hosted LLM APIs is costly, and the behavior of hosted models may change over time. Finally, the retrieval window $K = 1$ was selected on SWE-Bench Lite (Appendix D) and may not be optimal for other domains.

Broader impacts. By storing agent memory as plain-text, versioned files, GCC makes long-horizon agent behavior easier to inspect, audit, and hand over between agents and humans, which can improve the reliability and transparency of autonomous coding and research assistants. At the same time, more capable autonomous agents could be misused, for example to automate the development of malicious software. Persistent memory may also retain sensitive information encountered during a task (e.g., credentials or personal data) and stale or incorrect memory may mislead future agents. We recommend that deployments restrict access to the `.GCC/` workspace, redact sensitive content before committing it, and keep humans in the loop when merging agent-generated changes.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state the contributions: a Git-style context management framework (Section 2) and its empirical gains on SWE-Bench and BrowseComp-Plus (Section 4), supported by further comparisons and ablations in Appendices A–E.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Appendix G discusses limitations, including reliance on the backbone LLM, the scope of the evaluated domains, the increased number of tool calls, and the absence of error bars.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper does not include theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: Section 2 fully specifies the GCC file system and commands, and Section 3 specifies the benchmarks, LLM backbones, context budgets, retrieval window K, and baselines.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The GCC code is publicly available at `https://github.com/ImprintLab/git-context-controller`. All benchmarks used are publicly available.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 3 gives the benchmarks, context window, number of archives/branches, and baselines; Appendix D justifies the choice of K. GCC uses pretrained LLMs without any training.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: We report Pass@1 and resolution rates without error bars because repeating every experiment across all hosted LLM backbones is costly (Appendix G).

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Section 3.2 gives the context budgets; per-task API cost and token usage are reported in Table 2 and Appendices B and C.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We have reviewed the NeurIPS Code of Ethics; the work uses publicly available benchmarks and does not involve human subjects.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Appendix G discusses positive impacts (auditable and transferable agent memory) and negative impacts (misuse of more capable autonomous agents, sensitive information retained in persistent memory), together with mitigations.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: We release a context-management framework, not pretrained models or scraped datasets, so the work poses no high risk of misuse.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All benchmarks (SWE-Bench, SWE-Bench Lite, BrowseComp-Plus, LoCoMo, LongMemEval, LifeBench), baseline systems, and LLMs are cited, and are used in accordance with their respective licenses and terms of use.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.

- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The GCC code is released at `https://github.com/ImprintLab/git-context-controller` together with documentation of the file system and commands described in Section 2.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are a core component of the method: GCC is a context-management framework for LLM agents. The LLM backbones used (e.g., GPT-5, GPT-4.1, DeepSeek-V3.1, GLM-4.5-Air, Qwen3-235B-A22B, Claude 4 Sonnet) and how they are used are described in Sections 2 and 3.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.