Git Context Controller: Manage the Context of LLM-based Agents like Git

Junde Wu
University of Oxford
jundewu@ieee.org

Abstract

Large language model (LLM)-based agents have shown impressive capabilities by interleaving internal reasoning with external tool use. However, as these agents are deployed in long-horizon workflows, such as coding for a big, long-term project, context management becomes a critical bottleneck. We introduce Git-Context-Controller (GCC), a structured context management framework inspired by software version control systems. GCC elevates context from passive token streams to a navigable, versioned memory hierarchy. It structures agent memory as a persistent file system with explicit operations: COMMIT, BRANCH, MERGE, and CONTEXT, enabling milestone-based checkpointing, exploration of alternative plans, and structured reflection. Our approach empowers agents to manage long-term goals, isolate architectural experiments, and recover or hand off memory across sessions and agents. Empirically, agents equipped with GCC achieve state-of-the-art performance on the SWE-Bench-Lite benchmark, resolving 48.00% of software bugs—outperforming 26 competitive systems. In a self-replication case study, a GCC-augmented agent builds a new CLI agent from scratch, achieving 40.7% task resolution, compared to only 11.7% without GCC. The code is released at: `https://github.com/theworldofagents/GCC`

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 CLI-based usage is that sessions become increasingly slow and costly as context grows, since longer histories are passed as tokens. Yet closing a session and starting a new one typically erases the agent's memory of prior goals, user preferences, and task-specific instruction. As a result, users are forced to repeatedly "teach" the model from scratch across sessions. Current implementations rely on a few common strategies. The most straightforward is to truncate older context once the token limit is reached. While simple, this risks discarding important 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 todo-lists, as seen in Claude Code and Gemini CLI. These systems persist abstracted task state (e.g., via a single `memory.md`) and use summary-based anchors for future reasoning. However, relying on a simple compression means removing the fine-grained details, weakening the agent's ability to ground its actions in specific prior thoughts. Currently, context is 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 operations are triggered by the agent in response to its evolving internal state—supporting long-horizon planning, compositional reasoning, and reproducible workflows.

Such a design benefits in multiple ways:

- **Multi-level context retrieval**: Agents can access context at varying levels of detail, from high-level project plans to low-level OTA (Observation–Thought–Action) steps. The system enables seamless navigation across these layers, making it easy to trace and locate any point in the reasoning history. The agent can begin with a broad summary and drill down into fine-grained execution traces as needed.
- **Isolated exploration via branching**: Each branch acts as a safe workspace for the agent to explore new ideas, make mistakes, or iterate freely without affecting the main plan. This ensures the agent stays focused and avoids interference from side explorations, while maintaining the ability to return to the main reasoning flow at any time.
- **Cross-agent flexibility**: The system allows agents to operate seamlessly across sessions. There is no need to ”re-teach” the model when a new session begins. Another agent, based on a different LLM on a different machine, can also pick up exactly where the previous one left off with minimal overhead. Such a protocol helps smooth distribution and handover of agent-generated (vide-coded) codebases, much like how human developers collaborate through Git repositories.

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.
- Equipped LLM-based agents with GCC, it gets empirical SOTA Results on SWE-Bench, which outperforms 26 existing systems (open and commercial), achieving 48.00% resolution. We also conduct a case study in which a Claude-powered CLI agent, equipped with GCC, is tasked with building another CLI system from scratch. The GCC-augmented agent outperforms its non-GCC counterpart by a large margin on the SWEBench benchmark (40.7% vs. 11.7%), suggest a pathway toward autonomous agents capable of recursive self-improvement.

2 Method

The `Git-Context-Controller` (GCC) is 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. Inspired by version control systems like Git, GCC transforms the agent's ephemeral context into a persistent, navigable, and semantically meaningful workspace. Agents interact with the controller through commands such as `COMMIT`, `BRANCH`, `MERGE`, and `CONTEXT`, which manipulate a directory structure rooted at `.GCC/`. This structure includes global planning files (`main.md`), per-branch execution traces (`log.md`), milestone summaries (`commit.md`), and metadata (`metadata.yaml`) capturing architecture and file state.

The `Git-Context-Controller` organizes agent context into a structured directory rooted at `.GCC/`, with the following layout:

```
.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)
    |-- ...
```

Agents interact with the controller by calling designed commands include:

- `COMMIT <summary>` – called when the agent notices recent progress forms a coherent milestone (e.g., implementing a module, finishing a hypothesis test);
- `BRANCH <name>` – called when the agent wants to pursue an alternative approach without affecting current context (e.g., exploring a new API design or research path);
- `MERGE <branch name>` – called when a completed branch's results should be synthesized back into the main trajectory;
- `CONTEXT <options>` – called when the agent needs to retrieve project history, branch summaries, or fine-grained execution logs.

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.

In the following, we provide a detailed introduction to the structured file system and command interface of GCC.

2.1 GCC File System

The `Git-Context-Controller` (GCC) organizes agent memory 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.

`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`. Each command has a defined invocation format, behavioral trigger, and corresponding effect on the file system and agent memory. 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, it:

- Updates `commit.md` with a new entry containing:
  - The branch intent (inherited or re-iterated if needed from the `BRANCH` command);
  - A coarse-grained summary of progress up to the last commit, regenerated by combining the previous summary with the latest work;
  - A detailed description of what was achieved in this specific commit.
- Optionally prompts the agent to revise `main.md` if the project roadmap has shifted or the global plan has evolved.
- Finalizes the memory and code changes as a Git commit, using the agent-authored summary as the commit message.

This mechanism turns a loose sequence of OTA steps into a coherent and retrievable memory unit, enabling progress tracking and rollbacks.

`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.

When `BRANCH <name>` is issued:

- The controller creates an empty `log.md` file to track new OTA cycles specific to this branch.
- It initializes a new `commit.md`, prompting the agent to write an explanation of the branch's purpose and motivation.

This enables isolated reasoning, where experiments or alternative workflows are sandboxed from the mainline trajectory, yet remain fully inspectable and reversible.

`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.

The merge process involves:

- Updating `main.md` with a summary of the branch's outcome and its impact on the broader roadmap. If the branch contributes to future milestones, they are reflected accordingly.
- Merging `commit.md` entries from both branches under a unified structure. The updated file retains:
  - Optionally update the branch purpose;
  - Merge the target branch progress to the current one;
  - A detailed entry explaining the synthesis and outcome of the merge.
- Merging `log.md` files with origin tags (e.g., `== Branch A ==`) to preserve the traceability of OTA steps.
- Creating a new Git commit to checkpoint the unified memory state.

`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 calls `CONTEXT`, the controller first returns a `git status`-style snapshot containing:

- The project's purpose and milestone progress from `main.md`;
- A list of available branches.

To inspect a specific branch, the agent can issue:

- `CONTEXT --branch <branch>`, which returns:
  - The branch's purpose and progress summary from the last `commit.md`;
  - The latest 10 commits, each with a message and identifier. The agent can call `scroll_up` or `scroll_down` to view more entries.
- `CONTEXT --commit <hash>` to view a specific commit entry in full detail. Return the full `commit.md` content.
- `CONTEXT --log` to view the last 20 lines of `log.md`, with scrolling commands to traverse the full trace.
- `CONTEXT --metadata <segment>` to fetch the corresponding segment from `metadata.yaml`, such as `file_structure` or `env_config`.

This retrieval system gives agents structured, scoped access to their reasoning history.

3 Experiment

Datasets. To evaluate ours and competing methods, we use the widely adopted SWE-Bench benchmark Jimenez et al. (2024), which assesses the capability to resolve real-world software engineering issues. Each task in SWE-Bench involves generating a code patch to fix a specific bug described in the issue report. Our experiments primarily focus on the SWE-Benchlite subset swe (2024), a higher-quality and more self-contained collection of 300 tasks commonly used in recent evaluations.

Baselines. We compare GCC against 26 state-of-the-art agent-based systems. These baselines span both open-source and commercial tools, with closed-source methods marked using . For additional comparison, we also include a simple retrieval-augmented generation (RAG) baseline proposed in the original SWE-Bench paper Jimenez et al. (2024).

Metrics. We adopt the evaluation protocol from prior work Zhang et al. (2024), reporting the following key metrics: **(1) % Resolved** — the percentage of tasks successfully fixed by the system; **(2) Avg. Cost** — the average inference cost per task; and **(3) Avg. Tokens** — the average number of tokens consumed per query (input and output combined). We further assess patch localization accuracy through the metric **% Correct Location**, which measures how often a tool's patch modifies the same locations as the human-written ground truth. This metric is calculated at three levels of granularity: file, function, and line. A patch is considered correct if it modifies a superset of all relevant locations in the reference patch.

Table 1: 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.

Tool	LLM	% Resolved	Avg.	Avg.	% Correct Location		
			$ Cost	# Tokens	Line	Function	File
CodeStory Aide cod (2024)	GPT-4o+ Claude 3.5 S	129 (43.00%)	-	-	41.7%	58.7%	72.0%
Bytedance MarsCode Liu et al. (2024)	NA	118 (39.33%)	-	-	42.7%	58.0%	79.7%
Honeycomb hon (2024)	NA	115 (38.33%)	-	-	44.3%	57.0%	69.3%
MentatBot men (2024)	GPT-4o	114 (38.00%)	-	-	37.3%	53.3%	69.3%
Gru gru (2024)	NA	107 (35.67%)	-	-	38.3%	54.3%	75.0%
Isoform iso (2024)	NA	105 (35.00%)	-	41,963	38.7%	55.3%	72.0%
SuperCoder2.0 sup (2024)	NA	102 (34.00%)	-	-	41.7%	63.7%	65.7%
Alibaba Lingma Agent lin (2024)	GPT-4o+ Claude 3.5 S	99 (33.00%)	-	-	40.0%	58.7%	75.0%
Factory Code Droid fac (2024)	NA	94 (31.33%)	-	-	36.7%	55.7%	72.7%
Amazon Q Developer-v2 ama (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%)	$0.65	-	-	-	-
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 sim (2024)	GPT-4o	83 (27.67%)	$0.82	-	37.0%	54.0%	79.0%
IBM Research Agent-101 ibm (2024)	NA	80 (26.67%)	-	-	39.7%	56.7%	73.3%
OpenCSG StarShip ope (2024a)	GPT-4	71 (23.67%)	-	-	39.0%	61.7%	90.7%
Amazon Q Developer ama (2024)	NA	61 (20.33%)	-	-	34.0%	43.7%	71.7%
RepoUnderstander Ma et al. (2024)	GPT-4	64 (21.33%)	-	-	-	-	-
AutoCodeRover-v2 aut (2024)	GPT-4o	92 (30.67%)	-	-	35.0%	52.3%	69.3%
RepoGraph rep (2024)	GPT-4o	89 (29.67%)	-	-	36.7%	51.3%	71.0%
Moatless moa (2024)	Claude 3.5 S	80 (26.67%)	$0.17	-	38.7%	54.7%	78.7%
	GPT-4o	74 (24.67%)	$0.14	-	36.0%	52.0%	73.0%
OpenDevin+CodeAct v1.8 ope (2024b)	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 app (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-2	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%
Ours	Claude 3.5 S	144 (48.00%)	$2.77	569,468	44.3%	61.7%	78.7%

3.1 Results and Analysis

We report results on the SWE-Benchlite benchmark in Table 1, comparing GCC against a diverse set of 26 agentic coding systems. GCC achieves the highest resolution rate on SWE-Benchlite at **48.00%**, outperforming all previously reported systems, including both open and closed-source baselines. The next-best method, CodeStory Aide (43.00%), relies on a hybrid of GPT-4o and Claude 3.5 Sonnet, but does not release intermediate reasoning traces. Other high-performing closed-source agents such as ByteDance MarsCode (39.33%) and Honeycomb (38.33%) also fall short.

In terms of localization accuracy, GCC reaches **44.3% line-level**, **61.7% function-level**, and **78.7% file-level** correctness—consistently ranking among the top performers. For example, it outperforms the function-level accuracy of Lingma (58.7%) and matches or surpasses the line/file-level precision of Honeycomb, Sima, and ByteDance MarsCode. These results indicate that GCC not only solves more tasks, but also more accurately targets the correct edit regions. The retrieval-only RAG baselines remain far behind, resolving fewer than 5% of tasks and achieving poor localization performance. AgentLess improves upon this (32.00% resolved) but still lags behind agent-based approaches. These findings further underscore the advantage of GCC for complex software reasoning tasks.

3.2 Case Study of Self-evolving Agents: Reproduce CLI by CLI

In this case study, we examine a self-replicating experiment in which a Claude Code CLI, equipped with GCC, is tasked with implementing a new CLI system from scratch.

We examine three configurations on the SWEBench benchmark: (1) the original CLI system, (2) a second-generation CLI reproduced by the original CLI agent without GCC, and (3) a second-generation CLI reproduced by a GCC-augmented agent. All systems are evaluated on the same task distribution under consistent execution constraints.

As shown in Table 2, the original CLI resolves 72.7% of SWEBench tasks. However, when the same agent is prompted to reproduce this CLI from scratch without GCC, the resulting system performs drastically worse, resolving only 11.7% of tasks. In contrast, when equipped with GCC—and without access to the original CLI's design or source code, the agent's reproduction achieves a 40.7% resolution rate on SWEBench. This performance gain emerges despite both systems using the same language model and tool API, which demonstrates the difference lies not in capability, but in how that capability is scaffolded.

Table 2: SWEBench Task Resolution Rates Across CLI Variants

Setting	**Resolved (%)**
CLI	72.7
CLI-Produced CLI w/o GCC	11.7
CLI-Produced CLI w/ GCC	**40.7**

This result underscores the potential of self-evolving agents that capable of reproducing itself from scratch, and incrementally improving performance through iterative design. Such behavior hints at a developmental trajectory toward systems that can autonomously bootstrap increasingly sophisticated capabilities, an early glimpse into the mechanics of emergent superintelligence.

The remainder of this section analyzes two illustrative behaviors of agents in producing CLI: a commit-based modularization and branch-driven memory exploration, that surfaced spontaneously through the agent's interaction with the GCC protocol.

3.2.1 Commit Behavior: Implementing `write_file`

We observe that the GCC-powered CLI system spontaneously committed to implementing a `write_file` function during the construction of a child CLI. Before arriving at this implementation, the agent began reasoning about the limitations of transient output. In its early, non-GCC setup, commands returned results directly to the terminal via `stdout`, a pattern sufficient for simple, one-shot tasks but fragile for workflows requiring state persistence or multi-step composition. Once embedded in the structured memory environment of GCC, the agent began reflecting on richer scenarios that transforming a file and retrieving the result later and chaining outputs as inputs into downstream commands. Without any explicit prompting, its reasoning shifted from solving isolated tasks to considering the infrastructure needed for long-horizon reusable workflows. Given access to an organized context of prior experience, the agent inferred that a persistent file I/O mechanism would be essential, and proposed introducing a reusable utility function—`write_file(path, content)` to generalize file output operations across the CLI.

In its committing, the agent implemented the utility in a standalone module, `io.py`. It also generated a minimal test routine that wrote content to a temporary file, read it back, and confirmed correctness. Only after the test passed did the agent determine that the implementation was stable enough to warrant a versioned checkpoint. The `commit.md` was structurally recorded the contribution as:

```
## This Commit Contribution
Defines
↪  a reusable file output abstraction `write_file(path, content)` in `io.py
↪ `. Validated with a test to ensure correctness and future extensibility.
```

What is notable about this sequence is that it was not procedurally enforced. No rigid rule instructed the agent to write a test, validate correctness, and then commit. These behaviors emerged spontaneously from the framing structure of GCC itself. The system offered affordances: explicit commit boundaries and modular memory locations, and the agent internalized these affordances as cognitive norms. It came to understand that a commit contribution as a conceptually complete and stable architectural addition. In effect, the agent behaved as if it were operating within a disciplined software engineering workflow without having been explicitly told to do so.

3.2.2 Branching Behavior: Exploring the Design of History Processor

We observed that the CLI system, once equipped with GCC, began spontaneously reasoning about how historical context could evolve beyond basic textual logging. In its earlier, non-GCC form, the agent had been compressing prior interactions into a single `memory.md` file, a structure sufficient for tracking high-level progress, but lossy in detail and inadequate for fine-grained access. With the affordances provided by GCC, particularly the ability to branch architectural experiments, the agent began to reflect on what kind of memory processor could better support long-horizon reasoning and detailed recall. This reflection emerged organically, not from failure or instruction, but from the structural flexibility GCC introduced.

Rather than modifying the compressed memory system in-place, we find the agent invoked the `BRANCH` command to explore an alternate hypothesis that memory could be reimagined as a retriever-augmented knowledge base. In this new branch, it began prototyping a RAG-style memory layer, logging each OTA step as an atomic, vector-indexed unit to enable semantic querying and trace-level introspection. The new branch, titled `RAG-memory`, was initialized with a minimal `commit.md` defining its architectural intent:

```
Branch Purpose

Prototype a retriever-augmented memory system that indexes fine-grained
↪ OTA records to support semantic retrieval and long-horizon reasoning.
```

In this branch, the agent began prototyping a RAG-style memory module. Rather than compressing context into a single summary file, it logged each OTA block as an individual JSON entry. These records were embedded using a vector encoder and indexed to support semantic retrieval via a dedicated `retrieve_events(query)` function. To assess the effectiveness of this design, the agent integrated the RAG-based memory into its full reasoning loop and tested it on the SWE-Bench benchmark, aiming to determine whether fine-grained historical retrieval could enhance performance on complex, multi-step software engineering tasks.

After a round of experimentation, the agent itself concluded that the RAG-based approach introduced more drawbacks than benefits. It proved fragile—minor bugs in serialization or indexing logic led to unpredictable behaviors. It was computationally expensive, requiring non-trivial overhead to embed and store every intermediate step. It also underperformed in this implementation, the retriever-based memory yielded slower resolution times and lower task success rates than the simpler, compressed summary-based memory in the main branch. The agent documented these trade-offs and ultimately chose to abandon the RAG-based approach, reverting to the mainline memory system. Although the alternative was not retained in the final design, the process revealed an agent capable of exploring divergent ideas, testing them in isolation, and converging on a working solution only when sufficient empirical evidence supported it.

Such reflective behavior did not emerge in non-GCC environments, where agents tended to solve tasks in one shot, operating within narrow, ephemeral contexts. In contrast, GCC provided a structured space where architectural hypotheses could be explored safely—via branches—and evaluated on their own merits.

What makes this behavior remarkable is its spontaneity. The branching was not prompted. The comparison was not hard-coded. The decision to abandon the RAG memory system was not pre-scripted. These steps emerged organically from the agent's interaction with the affordances of the GCC. The agent behaved like a modular system architect—reasoning over its own workflows and encoding that reasoning structurally into the project itself.

Conclusion

In this work, we present Git-Context-Controller (GCC), a structured context management framework that equips LLM-based agents with version control-inspired operations to persist, organize, and retrieve memory across long-horizon workflows. By treating agent reasoning as a modular, evolving codebase, GCC enables disciplined behaviors such as milestone-based committing, architectural branching, and structured reflection. Empirical results on SWE-Bench-Lite show that GCC-equipped agents achieve state-of-the-art performance, and a self-replication case study demonstrates the emergence of recursive improvement. These results suggest that memory scaffolding not just model capability but a key to building autonomous, self-evolving agents.

References

Amazon q developer the most capable generative ai–powered assistant for software development. `https://aws.amazon.com/q/developer//`, 2024.

Appmap speedruns to the top of the swe bench leaderboard. `https://appmap.io/blog/2024/06/20/appmap-navie-swe-bench-leader/`, 2024.

Autocoderover autonomous software engineering. `https://autocoderover.dev/`, 2024.

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

Factory bringing autonomy to software engineering. `https://www.factory.ai/`, 2024.

The road to ultimate pull request machine. `https://gru.ai/blog/road-to-ultimate-pull-request-machine/`, 2024.

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

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.

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

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

Mentatbot: New sota coding agent, available now. `https://mentat.ai/blog/mentatbot-sota-coding-agent`, 2024.

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

Opencsg starship. `https://opencsg.com/product?class=StarShip/`, 2024a.

Opendevin: Code less, make more. `https://github.com/OpenDevin/OpenDevin/`, 2024b.

Repograph: Enhancing ai software engineering with repository-level code graph. `https://github.com/ozyyshr/RepoGraph`, 2024.

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

Supercoder. `https://superagi.com/supercoder/`, 2024.

Swe-bench lite. `https://www.swebench.com/lite.html`, 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.

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.

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

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://openreview.net/forum?id=VTF8yNQM66`.

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

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

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)*, pp. 28489–28503, 2025.

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

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.

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

A Appendix