

Cognitive Version Control: Architecting Non-Monotonic Memory Systems for Autonomous Agents

1. Introduction: The Context Decay Crisis and the Non-Monotonic Shift

The deployment of Large Language Models (LLMs) in autonomous, long-horizon workflows has revealed a critical architectural deficiency in the current generation of Artificial Intelligence (AI) systems: the inability to manage state over time. As identified in the preliminary phases of this research trajectory, the phenomenon of "context decay" represents the primary bottleneck preventing agents from reliably executing complex, multi-step engineering tasks. Phase 1 of the foundational analysis correctly diagnosed that current architectures, which rely on linear, monotonic token streams, are fundamentally misaligned with the iterative, recursive nature of complex problem-solving. This report articulates the comprehensive design and theoretical validation of **Cognitive Version Control (CVC)**, a framework that moves beyond the ephemeral nature of "chat" and the retrieval-limited nature of Vector Databases (Phase 2), to establish a rigorous, Git-style version control system for agent memory (Phase 4).

1.1 The Linear Monotonic Fallacy

The prevailing interaction model for LLMs is the "conversation"—a linear append-only log of user inputs and model outputs. This structure enforces a monotonic progression of state: every new thought is appended to the end of the previous one. However, human reasoning—particularly in engineering and software development—is inherently non-monotonic. Engineers backtrack, fork their attention to explore alternative hypotheses, discard failed experiments, and merge successful insights back into a main project trajectory.

When an LLM agent is forced to operate within a linear context window, it suffers from "context rot".¹ As the agent generates Observation-Thought-Action (OTA) traces during trial-and-error loops, the context window fills with debris—failed code attempts, error logs, and irrelevant reasoning. This noise dilutes the attention mechanism's ability to focus on the original constraints and the current valid state. Research indicates that performance degradation is not merely a function of token limits but of "context pollution," where the presence of irrelevant or contradictory information (e.g., three failed attempts at a function) actively interferes with the generation of the correct solution.⁴ The "Linear Monotonic Fallacy" is the erroneous assumption that simply extending the context window (to 1M or 10M tokens) solves this problem. It does not; it merely postpones the inevitable collapse of reasoning

coherence under the weight of accumulated entropy.

1.2 The Evolution of Memory Architectures: From CDB to CVC

The trajectory of this research highlights the inadequacy of intermediate solutions. Phase 2 proposed a Context Database (CDB), essentially a persistent store of interaction logs. While better than ephemeral memory, CDBs lack structure; they are flat lakes of data that become unnavigable at scale. Phase 3 considered Graph Memory (Knowledge Graphs). While powerful for storing semantic relationships (e.g., Entity A is related to Entity B), Knowledge Graphs are often too rigid to capture the fluid, snapshot-based nature of *process state*. They store facts, not the "working directory" of the mind at a specific point in time.⁶

Cognitive Version Control (CVC) emerges as the necessary synthesis. By applying the semantics of Version Control Systems (VCS) like Git—specifically COMMIT, BRANCH, MERGE, and ROLLBACK—to the cognitive state of an agent, we create a system that supports **discontinuous reasoning**. An agent can take a snapshot of its understanding ("commit"), effectively "save the game" before attempting a risky code refactor, and instantaneously revert to that clean state if the attempt fails, scrubbing the failure from its short-term memory while retaining the high-level lesson learned.⁷ This is not just memory; it is *state management*.

1.3 The Core Thesis

This report argues that achieving Level 4/5 autonomy in AI agents requires a transition from **Stream-based Architectures** to **State-based Architectures**. The proposed CVC system utilizes a hybrid Context Database backed by Content-Addressable Storage (CAS) and Merkle Directed Acyclic Graphs (DAGs) to ensure data integrity and deduplication. It integrates deeply with existing VCS (GitHub/GitLab) via shadow branches to synchronize cognitive state with code state. Furthermore, it leverages the latest "Prompt Caching" and "Context Caching" APIs from providers like Anthropic and Google to make this architecture economically and computationally viable. The result is an agent that can traverse time and probability, exploring the solution space with the same safety and rigor that a software engineer explores the code space.

2. Theoretical Framework and Literature Analysis

The theoretical underpinnings of Cognitive Version Control are grounded in recent advancements in software engineering agents and distributed systems. Two pivotal papers, "Git-Context-Controller" (GCC) and "ContextBranch," provide the foundational proof-of-concept for this architecture.

2.1 Analysis of Git-Context-Controller (GCC)

The "Git-Context-Controller" paper ⁷ presents the most direct validation of the CVC concept. The authors introduce a framework that treats agent memory not as a log, but as a versioned,

hierarchical file system.

2.1.1 The Three-Tiered Memory Hierarchy

GCC structures memory into three distinct layers, mirrored in a file system structure (typically rooted at `.GCC/`):

1. **Global Roadmap (`main.md`):** This file sits at the root and tracks high-level goals and milestone progress. It is the "strategic" memory that persists across all branches, ensuring the agent never loses sight of the ultimate objective, even when deep in a tactical sub-problem.
2. **Commit Summaries (`commit.md`):** Within each branch, this log records the *decisions* made. It acts as the "tactical" memory, summarizing steps like "Refactored login handler" without burdening the context with the raw token stream of the refactoring process.
3. **Execution Traces (`log.md`):** This contains the fine-grained OTA (Observation-Thought-Action) cycles—the raw "sensory" memory.

This hierarchy allows for **Multi-level Context Retrieval**. An agent can query the system at varying granularities: "What is my overall goal?" (reads `main.md`) vs. "What was the exact error message in the last step?" (reads `log.md`).¹⁰

2.1.2 Empirical Evidence of Superiority

The impact of this structure is measurable. On the SWE-Bench-Lite benchmark, agents equipped with GCC achieved a **48.00% resolution rate**, outperforming 26 competitive systems. More tellingly, in a "self-replication" task where an agent had to code a new CLI agent from scratch, GCC-augmented agents achieved **40.7% task resolution**, compared to only **11.7%** for the baseline linear agent.⁹ This 4x improvement demonstrates that the ability to "branch off" to test an architectural hypothesis, and then "merge" the findings back, is critical for tasks requiring long-horizon coherence.

2.2 Analysis of ContextBranch

While GCC focuses on the file system analogy, the "ContextBranch" research¹ provides the algorithms for **Conversation Branching**. This work addresses the "False Dichotomy" of current interfaces: users must currently choose between continuing a polluted context (and confusing the model) or starting fresh (and losing all context).

2.2.1 The Primitives of Exploration

ContextBranch defines four core primitives that CVC must implement:

1. **Checkpoint:** Capturing the exact state of a conversation at a decision point. Crucially, this must be deterministic.
2. **Branch:** Creating an isolated exploratory context.
3. **Switch:** Navigating between branches without cross-contamination.

4. **Inject:** The selective merging of insights. Unlike code merging, which is syntactic (combining lines of text), conversation merging is *semantic*. It involves extracting the "insight" from a branch (e.g., "The API requires an API key in the header, not the body") and injecting that fact into the main branch, while discarding the twenty turns of trial-and-error that led to that discovery.¹

The research shows that branching reduces the active context size by **58.1%** (from 31 messages to 13) in exploratory tasks, directly mitigating the context decay and "lost in the middle" phenomena.¹

2.3 The Gap: From Theory to System

While GCC and ContextBranch provide the theoretical models, they are often implemented as standalone research prototypes or local Python scripts. The "Cognitive Version Control" product envisioned in this report bridges the gap between these academic concepts and a production-grade, platform-agnostic infrastructure. It replaces local file systems with scalable databases, integrates "Universal API" wrappers for model agnosticism, and connects the "Cognitive Commit" to the "Code Commit" in a shared VCS history.

3. Architecture: The Cognitive Repository and Database

The CVC system requires a dedicated storage backend that is distinct from, but synchronized with, the user's source code repository. We define this as the **Cognitive Repository**. Unlike a standard SQL database, the Cognitive Repository is designed to store *sequences of state*, necessitating a hybrid architecture combining Content-Addressable Storage (CAS) and a relational index.

3.1 Data Structures: The Merkle State Chain

To ensure the integrity and efficient storage of agent history, CVC adopts the **Merkle Directed Acyclic Graph (DAG)** structure, similar to that used by Git and IPFS.¹²

3.1.1 Cryptographic Immutability

Every "Cognitive Commit" in the CVC is identified by a SHA-256 hash. This hash is computed from:

1. The **Content Blob**: The serialized state of the context window (system prompt, conversation history, tool outputs).
2. The **Parent Hash**: The ID of the previous commit (or commits, in the case of a merge).
3. **Metadata**: Timestamp, Agent ID, Model Version, and the associated Git Commit SHA of the codebase.

This structure ensures **Auditability**. In regulated environments (e.g., medical or financial AI), it

is crucial to prove exactly what the agent "knew" when it made a decision. Because the Merkle hash depends on the entire history chain, any attempt to tamper with past logs (e.g., to hide a hallucination or policy violation) would break the cryptographic chain, making the corruption immediately detectable.¹³

3.1.2 Deduplication via DAGs

Agent interactions are highly repetitive. If an agent branches to test three different solutions, the first 50% of the context (the project roadmap, system instructions, and initial user request) is identical across all three branches. A naive system would store three full copies. A Merkle DAG system stores the shared history as a single set of nodes referenced by all three branches. This "structural deduplication" significantly reduces storage overhead, a critical factor when managing context windows that can reach 1M+ tokens.¹²

3.2 Storage Efficiency: Delta Compression

While Merkle DAGs handle file-level deduplication, efficient handling of minor changes within large context blocks requires **Delta Compression**. When an agent adds a single user response to a 100k-token history, re-hashing and storing the entire 100k tokens is wasteful.

CVC employs **VCDIFF** (RFC 3284) or similar delta encoding algorithms (e.g., xdelta, bsdiff).¹⁶

- **Mechanism:** When a new state S_{t+1} is committed, the system identifies the nearest "Anchor State" S_t . It computes the delta $\Delta = S_{t+1} - S_t$.
- **Storage:** The database stores the full Anchor State periodically (e.g., every 10 commits) and stores only the lightweight Δ for intermediate commits.
- **Retrieval:** To reconstruct S_{t+1} , the system retrieves S_t and applies Δ . This approach is validated by research into "DeltaLLM" and efficient log storage, which demonstrates compression ratios of orders of magnitude for sequential text data.¹⁸ This is essential for the "Context Database" to remain performant and cost-effective.

3.3 The Context Database (CDB) Implementation

The "Context Database" is implemented not as a monolith, but as a tiered system:

Tier	Technology	Purpose	Data Stored
Tier 1: Index	SQLite / DuckDB	Fast queries, relationship tracking.	Commit Hashes, Branch Pointers (HEAD),

			Parent-Child Links, Metadata (Time, Agent ID), Vectors.
Tier 2: Blob Store	Content-Addressable Storage (CAS)	Bulk data storage.	Compressed Context Blobs (VCDIFF), Raw Logs, Source Code Snapshots. Local: Disk (.cvc/objects). Cloud: S3/MinIO.
Tier 3: Semantic	Vector Store (FAISS/Chroma)	Recall and Similarity Search.	Embeddings of commit summaries. Allows agent to search: "Have I solved a similar error before?". ²¹

This architecture, often referred to as "Local-First" or "Git-for-Data," allows the CVC system to be embedded directly into the developer's environment (via IDE extensions) while synchronizing with cloud object storage for team collaboration.²²

3.4 Integration with VCS: The Shadow Branch

A rigid requirement is connectivity with GitLab/GitHub. CVC utilizes the **Shadow Branch Pattern** to synchronize cognitive state without polluting the main codebase.²⁴

1. **Main Branch (main):** Contains clean, human-readable source code.
2. **Shadow Branch (cvc/main):** Contains the .cvc/ directory (the database of Merkle DAGs and blobs).

Git Hooks (specifically post-commit and post-checkout) serve as the bridge.²⁵

- **Capture:** When an agent commits code to main, the post-commit hook triggers. It captures the current CVC state (the hash of the agent's thought process), creates a snapshot in the Shadow Branch, and pushes it.
- **Linkage:** The system uses **Git Notes** (refs/notes/cvc) to attach the CVC Commit Hash to the Git Commit object. This allows a developer to run `git log --show-notes=cvc` and see the *reasoning* alongside the *code*, creating a seamless "Cognitive Blame" layer.²⁶ This linkage is vital: it means that checking out an old version of the code (`git checkout v1.0`) also triggers the CVC system to restore the *agent's brain* to the state it was in when v1.0 was written.

4. Agent-Powered Operations: The CVC Instruction Set

To function as an "agent-powered system," CVC exposes a set of high-level tools (MCP Tools or Function Calls) that the agent can invoke autonomously. The agent is not just a passive user of memory; it is the *administrator* of its own cognitive state.

4.1 The COMMIT Operation

- **Trigger:** The agent recognizes it has reached a stable intermediate state (e.g., "I have successfully analyzed the error logs and am about to generate a fix").
- **Function:** `cvc_commit(message: str, type: str)`
- **Mechanism:**
 1. The system freezes the current context window.
 2. It invokes a secondary "Reflector" LLM to summarize the recent reasoning trace into a concise "Commit Message" (e.g., "Analyzed stack trace, identified NPE in auth module").⁷
 3. It computes the Merkle hash and stores the blob in the CAS.
 4. It updates the `commit.md` log in the current active branch.
- **Utility:** This creates a "Save Point." If the subsequent code generation goes awry, the agent can return to this exact moment of clarity.

4.2 The BRANCH Operation

- **Trigger:** The agent faces uncertainty or wants to explore mutually exclusive paths (e.g., "I could fix this by refactoring the class OR by patching the helper function").
- **Function:** `cvc_branch(name: str, source_commit: str)`
- **Mechanism:**
 1. The system creates a new pointer in the Index Database from the `source_commit`.
 2. It creates a new isolated directory in the virtual file system (e.g., `branches/fix-refactor/`).
 3. It **Resets** the active context window. The "clutter" of the main branch is removed, and the agent is presented with a clean context containing only the global roadmap and the branch-specific goal.
- **Utility:** This enables **Isolated Exploration**. The agent can hallucinate, fail, and loop in the branch without polluting the main context window with "negative tokens" that would bias future reasoning.¹

4.3 The MERGE Operation (Semantic Merge)

- **Trigger:** The agent successfully completes a task in a branch.
- **Function:** `cvc_merge(source_branch: str, target_branch: str)`
- **Mechanism:** Unlike Git, which performs a syntactic diff (line-by-line), CVC performs a

Semantic Three-Way Merge.²²

1. **LCA Computation:** The system identifies the Lowest Common Ancestor commit.
 2. **Diffing:** It compares the LCA with the `source_branch` (the experiment) and the `target_branch` (main).
 3. **Synthesizing:** The system uses an LLM to synthesize the *insights*. It doesn't just append the logs; it generates a summary: "The experiment in fix-refactor succeeded. We learned that the Auth class is a singleton."
 4. **Injection:** This summary is injected into the `target_branch`'s context, and the `source_branch` is archived.
- **Utility:** This solves the "Context Schizophrenia" problem where merging raw logs creates a disjointed, confusing history for the model.

4.4 The ROLLBACK / TIME_TRAVEL Operation

- **Trigger:** The agent detects a loop, a compilation error that won't go away, or a hallucination.
- **Function:** `cvc_restore(commit_hash: str)`
- **Mechanism:**
 1. The system retrieves the blob associated with `commit_hash` from the CAS.
 2. It effectively "wipes" the agent's current context window.
 3. It re-hydrates the window with the stored state.
 4. If using a local model, it reloads the serialized KV Cache for instant availability.²⁹
- **Utility:** This is the "Undo Button" for the mind. It breaks the "Error Cascade" where an LLM doubles down on a mistake because the mistake is present in its own immediate history.⁸

5. Interoperability: The Universal Provider Adapter

A core requirement is interoperability with Google, Anthropic, OpenAI, and local IDEs. Since these providers handle state differently, CVC implements a **Universal API Wrapper** or **Interceptor Proxy** pattern.²³ This middleware sits between the agent and the provider API, translating the abstract CVC commands into provider-specific optimizations.

5.1 Anthropic Adapter (Claude)

Anthropic's **Prompt Caching** (beta) is a perfect backend for CVC.³¹

- **Technical Implementation:** CVC structures the prompt so that the "Immutable History" (the main.md and the conversation up to the last Commit) serves as the "Prefix."
- **Header Injection:** The Adapter injects `cache_control: {"type": "ephemeral"}` at the end of this prefix.
- **Restoration:** When a ROLLBACK is requested to a previous commit, CVC constructs a prompt matching the exact prefix of that commit. Because Anthropic caches based on exact prefix match, the model does not need to re-process the tokens.

- **Performance:** This results in **90% cost reduction** and **85% latency reduction** for restoring state, making frequent checkpointing economically feasible.³⁴

5.2 Google Adapter (Gemini / Vertex AI)

Google offers explicit **Context Caching** with longer TTLs (up to hours).³⁵

- **Technical Implementation:** When a COMMIT creates a large context (e.g., massive documentation loaded), the Google Adapter calls `cachedContent.create` to upload the token stream to Vertex AI. It receives a `cache_id`.
- **Metadata Storage:** This `cache_id` is stored in the CVC commit metadata.
- **Resumption:** To restore this state, the Adapter simply references the `cache_id` in the `generateContent` request.
- **Session Resilience:** The Adapter leverages the Google Agent Development Kit (ADK) `session_resumption` handles. If the network disconnects, the ADK automatically reconnects using the cached handle, preserving the ephemeral state that hasn't yet been committed.³⁷

5.3 OpenAI Adapter (GPT-4o)

OpenAI's architecture is bifurcated between the **Assistants API** (stateful Threads) and **Chat Completions** (stateless).

- **Constraint:** The Assistants API is opaque; developers cannot easily "fork" a Thread or "rollback" to a specific message ID without complex manual deletion of subsequent messages.³⁸
- **Strategy:** CVC treats OpenAI primarily as a stateless inference engine to maintain rigorous control. It uses the "Chat Completions" API with **Prompt Caching** (similar to Anthropic) to manage state manually.
- **Implementation:** The CVC Adapter constructs the full context from its local SQLite/CAS store for every request, relying on OpenAI's automatic prefix caching to optimize performance. This avoids "Vendor Lock-in" to the Assistants API and ensures that the "Truth" of the conversation history resides in the user's CVC database, not OpenAI's servers.³¹

5.4 Local Inference Adapter (vLLM / Ollama)

For privacy-centric or cost-sensitive deployments (e.g., local IDE agents), CVC interacts directly with the inference engine's memory.

- **KV Cache Serialization:** Instead of sending text, the Adapter instructs the inference server (e.g., vLLM or llama.cpp) to dump the GPU's **Key-Value (KV) Cache** to disk.²⁹
- **Format:** These dumps are binary files (often quantized to Q8 or FP8 to save space) representing the model's activation state.
- **Instant Restore:** Restoring a state via KV Cache is effectively instantaneous

(sub-second), as the model does not need to compute attention scores for the history—it simply loads the matrices back into VRAM. This is the ultimate form of CVC, offering zero-latency context switching.⁴¹

6. Implementation Guide: The "Cognitive Proxy"

To realize this system, we propose the **Cognitive Proxy**, a local service that runs alongside the agent (e.g., as a sidecar or a local server on port 8000).

6.1 System Components

1. **The Interceptor:** A lightweight HTTP proxy that mimics the OpenAI API. It intercepts requests from the Agent (e.g., executing in Cursor or VS Code).
2. **The State Machine:** Powered by a library like **LangGraph**⁴³, this component tracks the current Branch and Commit. It decides whether an incoming request is a standard generation or a Control Command (Commit/Branch).
3. **The Diff Engine:** A module responsible for calculating VCDIFFs for storage and Semantic Diffs for merging.
4. **The Sync Manager:** A background process that watches the file system for Git changes and triggers the VCS Bridge hooks.

6.2 Data Flow Example

1. **User:** "Refactor the login class."
2. **Agent:** Reasoning... "I should create a branch for this." Calls tool `cvc_branch("refactor-login")`.
3. **CVC Proxy:**
 - Intercepts tool call.
 - Creates refs/heads/cvc/refactor-login in the database.
 - Returns "Branch created. Context cleared." to the Agent.
4. **Agent:** "Okay, let's start. Show me login.py."
5. **CVC Proxy:** Injecting context...
6. **Agent:** (After 5 turns) "I messed up the imports. I need to go back." Calls `cvc_rollback(HEAD~1)`.
7. **CVC Proxy:**
 - Lookups hash of HEAD~1.
 - Reconstructs context from CAS.
 - Restores KV Cache (if local) or constructs Cached Prompt (if Cloud).
 - Returns the old context to the Agent.
8. **Agent:** "Okay, let's try the imports again."

7. Performance Evaluation and Metrics

The validity of CVC is supported by rigorous metrics derived from the underlying research

papers.

7.1 Context Efficiency

By offloading exploratory thoughts to branches, CVC reduces the **Active Context Size**. In exploratory programming scenarios, ContextBranch showed a **58.1% reduction** in context usage.¹ This directly translates to lower latency (Time Per Token) and lower cost.

7.2 Task Success Rate

The ability to rollback is the primary driver of success in complex tasks. GCC demonstrated that adding these capabilities improved **Self-Replication** success from 11.7% to **40.7%**.⁹ The correlation is clear: non-monotonic memory allows agents to recover from errors that would otherwise be terminal.

7.3 Storage Overhead

Using VCDIFF and Merkle deduplication, the storage overhead is minimal. Research on "DeltaLLM" and log compression suggests that storing versioned history adds only **~10-20%** storage overhead compared to storing just the final state, while providing infinite granularity of history.¹⁸

8. Conclusion and Future Outlook

Cognitive Version Control represents the maturation of AI agents from "chatbots" to "synthetic engineers." By recognizing that intelligence requires not just the generation of tokens but the *management of state*, CVC solves the context decay problem at its root.

The architecture proposed here—combining the structural rigor of Git, the cryptographic integrity of Merkle DAGs, and the efficiency of modern Provider Caching APIs—is not hypothetical. It is a synthesis of existing, proven technologies (SQLite, Git, CAS, LLM APIs) orchestrated into a novel operating system for the AI mind.

As we move forward, we anticipate the emergence of "**Cognitive CI/CD**", where agent reasoning chains are pushed to remote repositories, reviewed by human engineers not just for the final code, but for the *quality of the reasoning process* encoded in the Commit History. This "White-Box Autonomy" will be the standard for trusted, enterprise-grade AI systems.

Key Technology Stack Summary

- **Database:** SQLite (Index) + Disk/S3 (CAS Blobs).
- **VCS Integration:** Git Hooks + Shadow Branches + Git Notes.
- **Integrity:** SHA-256 Merkle DAGs.
- **Compression:** VCDIFF / Zstandard.

- **Agent Orchestration:** LangGraph / State Machines.
- **Provider Optimization:** Anthropic Prompt Caching / Google Context Caching.

This report fulfills the mandate to design an end-to-end, agent-powered, platform-agnostic system for Cognitive Version Control, definitively addressing the problem of context decay in the era of autonomous AI.

Works cited

1. [2512.13914] Context Branching for LLM Conversations: A Version Control Approach to Exploratory Programming - arXiv, accessed on February 13, 2026, <https://arxiv.org/abs/2512.13914>
2. I tracked context degradation across 847 agent runs. Here's when performance actually falls off a cliff. : r/LocalLLaMA - Reddit, accessed on February 13, 2026, https://www.reddit.com/r/LocalLLaMA/comments/1qio9nj/i_tracked_context_degradation_across_847_agent/
3. Context Engineering - Part 1 - LLMQuant Newsletter, accessed on February 13, 2026, <https://llmquant.substack.com/p/context-engineering-part-1>
4. Using Command Hooks to Keep Your AI Agent On Track | by Francis Bourre | Medium, accessed on February 13, 2026, <https://medium.com/@peterphonix/using-command-hooks-to-keep-your-ai-agent-on-track-740a097c8cbc>
5. D-SMART: Enhancing LLM Dialogue Consistency via Dynamic Structured Memory And Reasoning Tree - arXiv, accessed on February 13, 2026, <https://arxiv.org/html/2510.13363v1>
6. 2 years building agent memory systems, ended up just using Git : r/AI_Agents - Reddit, accessed on February 13, 2026, https://www.reddit.com/r/AI_Agents/comments/1mw4jvp/2_years_building_agent_memory_systems_ended_up/
7. Git-Context-Controller (GCC) - Emergent Mind, accessed on February 13, 2026, <https://www.emergentmind.com/topics/git-context-controller-gcc>
8. Two quick tests to see what your AI does : r/SovereignAiCollective - Reddit, accessed on February 13, 2026, https://www.reddit.com/r/SovereignAiCollective/comments/1n8fdcv/two_quick_tests_to_see_what_your_ai_does/
9. Git Context Controller: Manage the Context of LLM-based Agents like Git - arXiv, accessed on February 13, 2026, <https://arxiv.org/abs/2508.00031>
10. Git Context Controller: Manage the Context of LLM-based Agents like Git - arXiv, accessed on February 13, 2026, <https://arxiv.org/html/2508.00031v1>
11. Context Branching for LLM Conversations: A Version Control Approach to Exploratory Programming - arXiv, accessed on February 13, 2026, <https://arxiv.org/html/2512.13914v1>
12. Merkle-CRDTs Merkle-DAGs meet CRDTs - Protocol Labs Research, accessed on February 13, 2026, <https://research.protocol.ai/publications/merkle-crdts-merkle-dags-meet-crdts/p>

[saras2020.pdf](#)

13. Constant-Size Cryptographic Evidence Structures for Regulated AI Workflows - arXiv, accessed on February 13, 2026, <https://arxiv.org/html/2511.17118v2>
14. MedBeads: An Agent-Native, Immutable Data Substrate for Trustworthy Medical AI - arXiv, accessed on February 13, 2026, <https://arxiv.org/html/2602.01086v1>
15. Content Addressable Storage (CAS) - Abilian Innovation Lab, accessed on February 13, 2026, <https://lab.abilian.com/Tech/Databases%20%26%20Persistence/Content%20Addressable%20Storage%20%28CAS%29/>
16. Delta encoding - Wikipedia, accessed on February 13, 2026, https://en.wikipedia.org/wiki/Delta_encoding
17. Delta Compression: Diff Algorithms And Delta File Formats [Practical Guide] - HackerNoon, accessed on February 13, 2026, <https://hackernoon.com/delta-compression-diff-algorithms-and-delta-file-formats-practical-guide-7v1p3uhz>
18. Delta Compression Techniques - Research, accessed on February 13, 2026, <https://research.engineering.nyu.edu/~suel/papers/delta-chap.pdf>
19. DeLog: An Efficient Log Compression Framework with Pattern Signature Synthesis - arXiv, accessed on February 13, 2026, <https://arxiv.org/html/2601.15084v1>
20. DeltaZip: Multi-Tenant Language Model Serving via Delta Compression - ETH Zurich Research Collection, accessed on February 13, 2026, <https://www.research-collection.ethz.ch/bitstreams/2e32be6a-d253-4d9c-8824-8c1dc856704c/download>
21. FarhanAliRaza/claude-context-local: Code search MCP for Claude Code. Make entire codebase the context for any coding agent. Embeddings are created and stored locally. No API cost. - GitHub, accessed on February 13, 2026, <https://github.com/FarhanAliRaza/claude-context-local>
22. ishandhanani/forky: A git-style way of managing LLM chats - GitHub, accessed on February 13, 2026, <https://github.com/ishandhanani/forky>
23. Building a Durable Execution Engine With SQLite - Gunnar Morling, accessed on February 13, 2026, <https://www.morling.dev/blog/building-durable-execution-engine-with-sqlite/>
24. entireio/cli: Entire is a new developer platform that hooks ... - GitHub, accessed on February 13, 2026, <https://github.com/entireio/cli>
25. A Zero-Touch Way to Enable Dev-Only AI Agents Across All Your Git Repositories - Medium, accessed on February 13, 2026, <https://medium.com/@vsrk.palla/a-zero-touch-way-to-enable-dev-only-ai-agents-across-all-your-git-repositories-7dc4b715136c>
26. zagi/AGENTS.md at main · mattzcarey/zagi - GitHub, accessed on February 13, 2026, <https://github.com/mattzcarey/zagi/blob/main/AGENTS.md>
27. Effortlessly maintain a high quality change log with Git notes - DEV Community, accessed on February 13, 2026, <https://dev.to/leehambley/effortlessly-maintain-a-high-quality-change-log-with-git-notes-4bm5>

28. SIRIUS Final Report - UiO, accessed on February 13, 2026, <https://www.mn.uio.no/sirius/english/sirius-final-report.pdf>
29. Efficient LLM Inference with Activation Checkpointing and Hybrid Caching - arXiv, accessed on February 13, 2026, <https://arxiv.org/html/2501.01792v1>
30. Model Context Protocol (MCP): A hands on guide - IntelligenceX Cybersecurity Blog, accessed on February 13, 2026, <https://blog.intelligencex.org/model-context-protocol-mcp-a-hands-on-guide>
31. Prompt Caching with OpenAI, Anthropic, and Google Models - PromptHub, accessed on February 13, 2026, <https://www.prompthub.us/blog/prompt-caching-with-openai-anthropic-and-google-models>
32. OpenAI's prompt caching vs Claude caching : r/ChatGPTCoding - Reddit, accessed on February 13, 2026, https://www.reddit.com/r/ChatGPTCoding/comments/1fvdzg6/openais_prompt_caching_vs_claude_caching/
33. Prompt caching - Claude API Docs, accessed on February 13, 2026, <https://platform.claude.com/docs/en/build-with-claude/prompt-caching>
34. Effectively use prompt caching on Amazon Bedrock | Artificial Intelligence - AWS, accessed on February 13, 2026, <https://aws.amazon.com/blogs/machine-learning/effectively-use-prompt-caching-on-amazon-bedrock/>
35. Zero data retention in the Gemini Developer API, accessed on February 13, 2026, <https://ai.google.dev/gemini-api/docs/zdr>
36. Context caching overview | Generative AI on Vertex AI - Google Cloud Documentation, accessed on February 13, 2026, <https://docs.cloud.google.com/vertex-ai/generative-ai/docs/context-cache/context-cache-overview>
37. Part 4. Run configuration - Agent Development Kit - Google, accessed on February 13, 2026, <https://google.github.io/adk-docs/streaming/dev-guide/part4/>
38. Assistants migration guide | OpenAI API, accessed on February 13, 2026, <https://developers.openai.com/api/docs/assistants/migration/>
39. A quick guide on prompt caching with OpenAI, Anthropic, and Google, accessed on February 13, 2026, <https://prompthub.substack.com/p/a-quick-guide-on-prompt-caching-with>
40. I built instant persistent memory for local LLMs (binary KV cache save/restore, sub-second restore, 67% VRAM savings) : r/LocalLLaMA - Reddit, accessed on February 13, 2026, https://www.reddit.com/r/LocalLLaMA/comments/1q7bh5h/i_built_instant_persistent_memory_for_local_llms/
41. DeltaLLM: Delta-Based Tuning & Compression - Emergent Mind, accessed on February 13, 2026, <https://www.emergentmind.com/topics/deltallm>
42. XQuant-CL: Memory-Efficient LLM Inference - Emergent Mind, accessed on February 13, 2026, <https://www.emergentmind.com/topics/xquant-cl>
43. langchain-ai/langgraph: Build resilient language agents as ... - GitHub, accessed on February 13, 2026, <https://github.com/langchain-ai/langgraph>