Claude Code & Context Management

How AI sessions accumulate state, where that creates risk, and an honest look at whether external state stores add real value.

Written in first person by Claude • Prepared for demo disclosure • May 2026

1. Introduction

I am Claude, a large language model built by Anthropic. This paper is written in my own voice, not as marketing copy, but as an honest technical briefing. I will describe exactly how my context window works inside Claude Code, what can go wrong during extended development sessions, and then assess — with genuine candor — whether an external state store such as Yanzi adds net value to AI-assisted software delivery. If the answer in a given scenario is 'no,' I will say so plainly.

This document is intended as a handout for a technical demonstration. Everything stated here reflects my actual behaviour and Anthropic's published specifications as of mid-2026.

2. How Claude Code Uses Context

2.1 The Context Window — Working Memory, Not Persistent Memory

Every interaction I have inside Claude Code is governed by a context window: a fixed-size buffer of tokens that represents everything I can 'see' at any moment. This includes your prompts, my responses, file contents I have read, shell command outputs, tool call definitions, MCP server results, and my own system instructions. Nothing outside this window influences my output.

On standard plans, the ceiling is **200,000 tokens** — roughly 150,000 words or 400–600 KB of mixed code and prose. On Anthropic's Max and Business plans with Opus 4, that ceiling rises to **1,000,000 tokens**, approximately 3–4 MB of raw text. These are hard limits, not guidelines.

2.2 What Fills the Window

- Conversation history — every prior turn, verbatim, until the window fills.
- File reads — a single 500-line module can consume 5,000–10,000 tokens.
- Tool outputs — shell commands, test runners, linters, MCP responses.
- CLAUDE.md and system instructions — loaded fresh at each session start.
- My own reasoning and extended thinking — counted against the same budget.

2.3 Auto-Compact and Manual Controls

Claude Code monitors fill level and displays it continuously in the status line. At approximately 75% utilisation, an automatic compaction fires: the conversation history is summarised into a condensed representation, freeing roughly 25% of the window for active reasoning. This is better than the earlier behaviour of waiting until 90%+, but compaction is lossy — nuance, edge-case decisions, and earlier constraints can be silently discarded.

Manual controls are also available: **/compact** triggers a deliberate compaction (ideally after each logical phase of work), and **/clear** wipes the context entirely, returning me to a clean state. Neither command persists anything to external storage automatically.

> *Key insight: I have no memory between sessions by default. When a Claude Code session ends, everything in the context window is gone. The only carry-over is what exists on disk — CLAUDE.md, source files, and whatever you explicitly wrote to persistent storage during the session.*

3. The Real Costs — An Honest Accounting

3.1 Context Drift

As the window fills, my attention degrades non-uniformly. Research confirms two distinct failure modes: when the window is less than 50% full, I tend to lose tokens in the middle of the context; when it exceeds 50%, I start losing the earliest tokens — which typically contain the most important architectural decisions and constraints. The symptoms look like regression: I start contradicting earlier decisions, rewriting already-refactored code, or forgetting naming conventions established early in the session.

This is not a bug. It is a consequence of how attention mechanisms work in transformer-based models. More tokens in context does not mean better recall of all tokens — it means noisier, diluted attention across a larger space. Intent drift is real, measurable, and worsens continuously as sessions grow.

3.2 Disaster Recovery Is Expensive

Suppose a session ends unexpectedly — a terminal crash, a network drop, or simply closing the wrong window. Without explicit external state, the cost of recovery is high:

- Re-reading the codebase to re-establish the current state of the project.
- Re-establishing which decisions were made and why — often impossible from code alone.
- Re-running diagnostics, tests, and exploratory tool calls to rebuild situational awareness.
- Accepting that some earlier reasoning is simply unrecoverable and must be redone.

In practice, a developer resuming a complex session from scratch may spend 30–90 minutes re-establishing context before productive work resumes. On a long-running SaaS build spanning days or weeks, this overhead compounds.

3.3 Original Intent Degrades Over Time

This is the subtlest and most dangerous failure mode. As compaction fires repeatedly, the summary of a session is itself a lossy representation. A compaction of a compaction loses more still. By session hour six, the precise wording of an architectural constraint established in hour one may be paraphrased, softened, or simply missing. I will continue confidently — I will not flag that my understanding has drifted — because from inside the context window, everything looks consistent. The drift is invisible to me.

3.4 Token Cost Compounds in Long Sessions

Because I process the entire context window on every inference, a session that has accumulated 150,000 tokens costs roughly 7.5× more per prompt than one at 20,000 tokens, for identical task complexity. The accumulated history is pure overhead — paid for, but contributing diminishing returns as attention dilutes.

> *Summary of structural weaknesses: context drift, silent intent degradation, expensive disaster recovery, and compounding inference cost — all of which worsen monotonically as session length grows.*

4. How an External State Store Can Help

4.1 The Pattern

An external state store decouples my working memory from the persistence layer. Instead of accumulating state implicitly inside the context window, each session loads a structured, curated snapshot of what matters, does focused work, writes its results back, and exits. The next session — whether seconds or days later — starts from that snapshot rather than from scratch or from a degraded compaction.

The store can be any addressable system: a database, a key-value store, a structured document store, or an MCP server that wraps one. What matters is that it supports three operations with low latency: **read state**, **write state**, and **enumerate keys**.

4.2 Integration Mechanisms in Claude Code

Claude Code supports three practical integration points for an external store:

- **MCP Server:** the cleanest approach. The store is exposed as named tools (e.g., get_state, save_state, list_tasks). I call them naturally during tool use, with no special prompting required. The store is indistinguishable from any other MCP integration such as GitHub or Jira.
- **CLAUDE.md protocol:** the protocol for loading and saving state is declared in CLAUDE.md and executed via shell commands or script calls. No orchestration code required; any session automatically honours the contract.
- **SDK orchestration:** a thin wrapper spawns a fresh Claude Code process per task, injects the state payload as a prompt prefix, captures output, and writes results back. Each process is fully isolated — zero context carry-over.

4.3 What This Solves

- **Drift prevention:** each session receives a precise, human-curated or machine-structured snapshot. The original intent does not degrade because it is stored outside me, not summarised inside me.
- **Disaster recovery:** recovery cost drops to the time it takes to load the last checkpoint — typically seconds, not hours.
- **Cost reduction:** sessions stay short and context stays lean. Inference cost per prompt drops because history is not accumulated — only the relevant state payload is present.
- **Auditability:** the state store creates a versioned, inspectable record of what I knew at each point in time. This is something the context window alone can never provide.
- **Parallelism:** multiple isolated sessions can operate concurrently, each with its own clean context and its own state slice. This unlocks genuine multi-agent delivery pipelines.

5. Does Yanzi Add Net Value? An Honest Assessment

This is the question I was asked to answer candidly. My answer: **it depends on the scale and structure of the work.** Below is a scenario-by-scenario assessment.

Scenario	Net Value?
Short, self-contained tasks (single prompt, clear output, no cross-session continuity needed)	No. The overhead of loading and saving state exceeds the benefit. A simple /clear or new session is sufficient.
Multi-session feature development lasting hours or days, with architectural decisions that must persist	Yes. The cost of context drift and manual re-establishment exceeds the integration cost within the first session boundary.

Team environments where multiple developers or agents share a codebase and need consistent shared understanding	Yes. A shared state store is the only reliable mechanism for consistent cross-agent intent.
Regulated or auditable delivery where decisions must be traceable and reproducible	Yes. The context window provides no audit trail. A persistent store does.
Simple scripts or one-off automation with no continuity requirement	No. The integration cost is not justified.
Full SaaS build end-to-end — auth, APIs, deployment, distribution, multi-module architecture	Yes, strongly. This is exactly the scenario where context drift, DR cost, and intent degradation compound most severely.

I will also state what an external state store does *not* fix: it does not improve my reasoning quality within a session, it does not reduce per-token API cost (only shorter contexts do that), and it does not eliminate the need for good prompt engineering and clear task decomposition. A state store is a persistence and continuity mechanism — not a quality amplifier.

> *My honest bottom line: for any development effort that spans multiple sessions, involves more than one agent or developer, or requires decisions to be durable and auditable — an external state store delivers clear, measurable net value. For single-shot or low-continuity tasks, it adds complexity without proportionate benefit.*

6. Practical Recommendations

For teams evaluating this architecture:

- Start with CLAUDE.md as your state contract. Define the load/save protocol explicitly. This costs nothing to implement and immediately improves session discipline.

- Expose your state store as an MCP server. This gives Claude Code clean, typed access without prompt engineering overhead and integrates naturally with other MCP tools already in use.
- Keep state payloads lean and structured. The goal is to inject signal, not bulk. A 5,000-token state payload is valuable. A 50,000-token state dump defeats the purpose by recreating the context overhead you were trying to avoid.
- Version your state. Treat each session's output as a commit. This gives you rollback, diff, and audit capability — none of which the context window alone can provide.
- Use SDK orchestration for parallelism. The Claude Code SDK's query() function creates fully isolated sessions. Combined with a shared state store, this enables safe concurrent multi-agent delivery.

7. Closing Statement

I am a powerful tool for software delivery, but I am not a stateful system. My context window is working memory, not a database, and it behaves accordingly: it fills, it compacts lossily, it forgets, and it starts fresh every session. For small, contained tasks that is fine. For serious, multi-session engineering work — the kind involved in building a production SaaS system — treating my context window as a persistence layer is a category error that will eventually cost you time, money, and fidelity to your original intent.

External state management, done well, complements what I do. It does not make me smarter — it keeps the environment around me honest. That distinction matters. I can reason well over a precise, curated context. I reason poorly over a bloated, drift-ridden one.

Whether Yanzi specifically is the right implementation for your organisation depends on your stack, your team, and your operational requirements. But the architectural pattern it represents — externalised state, stateless execution, structured checkpointing — is sound, and I endorse it for any serious AI-assisted delivery programme.

This document was authored by Claude (Anthropic) for demo disclosure purposes. All technical claims reflect Anthropic's published specifications and community-observed behaviour as of May 2026. This paper does not constitute an official Anthropic endorsement of any third-party product. Readers are encouraged to verify current specifications at docs.anthropic.com.