Context Engineering Field Guide

Engineer the LLM's working RAM

Jounes (from the lab)

2026

Context Engineering Field Guide

Jounes (from the lab)

2026

Evict: Clearing What the Session No Longer Needs

- what prompt soup actually contains
- compact vs. subagent-offload
- truncation policies for tool outputs
- the over-eviction failure mode
- worked example: a subagent dispatch and return
- eviction diagnostic

Sandwich Architecture

- the working vs. belonging gap
- the three slices
- why direction matters
- the naive single-prompt failure
- worked example: a sandwich, end to end
- sandwich architecture diagnostic

Orchestrator and IC Roster

- the main session as router, not editor
- subagent as context isolation, not parallelism
- the lead tier and the IC roster
- the dispatch/return protocol
- model selection per role
- worked example: one task, three dispatches
- orchestrator and IC roster diagnostic

Hook-Enforced Discipline

- why prompts drift and hooks don’t
- the four hook types
- no manual invocation required
- what to enforce with hooks
- the kill-switch pattern
- worked example: a hook, on disk
- hook-enforcement diagnostic

Eval-Driven Context Tuning

- the feedback loop that most teams skip
- what to measure

Context Is the New Code

You switched from GPT-4 to the newest model. The sessions got slightly faster. The output looked slightly better. And then, two weeks in, you hit the same wall you always hit: the agent forgets, drifts, thrashes, and dies mid-task.

The bottleneck was never the model. It was what you put in front of it.

---

the shift that snuck up on everyone

In 2009, Patrick Debois asked “what if ops looked more like dev?” and it cracked open DevOps. In 2025, the analogous question surfaced quietly across thousands of engineering teams: what if context is the code? (source: Patrick Debois (Tessl), “Context Is the New Code”)

That’s not a metaphor. It’s a description of where the work actually lives now. When you build with an AI coding agent, you barely touch the source files directly. You instruct. You describe. You write `CLAUDE.md` files (the persistent instruction file that tells an AI coding agent how to behave in your project) and system prompts and skill definitions. Those artifacts, not the code they generate, are what you maintain, version, and debug. The code becomes the artifact of the context, not the primary work product.

Context engineering is the discipline of designing, maintaining, and optimizing the information state you deliver to an AI agent so it can reason accurately and act reliably.

Notice what that definition does not say. It says nothing about which model you use. It says nothing about your prompt style or your preferred syntax for bullet points. Context engineering is about the complete state the agent receives: not just the system prompt, but also tool outputs, injected files, conversation history, and the structural decisions about what to include, what to compress, and what to discard.

The model is the engine. Context is the fuel. You cannot do much about the engine. You can control the fuel. Most teams have spent two years tuning the engine and almost no time engineering the fuel. (source: Patrick Debois (Tessl), “Context Is the New Code”)

---

why prompt-only thinking caps out

Most practitioners arrive at context engineering through failure, not foresight. They start with a system prompt. The prompt grows. Rules get added. Edge cases get enumerated. The file gets long. The agent still drifts. More rules go in. The file gets longer. Nothing gets better. Sometimes things get worse.

This is the prompt-only ceiling, and it hits every project that advances beyond trivial tasks.

The failure is structural. A system prompt is a static artifact injected at session start. As the session runs, conversation turns accumulate, tool outputs pile up, and the static instructions from the start of the session compete with the live noise of the middle. The model's attention isn't uniform across the context window; recent tokens carry more practical weight than early ones. Your carefully written rules, front-loaded in the session, gradually lose the competition.

The survey evidence is blunt: Dex Horthy, in his late-2025 lecture, cites a survey of 100,000 developers showing that most AI-assisted coding on complex, brownfield codebases results in rework and churn. Greenfield projects with narrow scope work better; the moment you're navigating a real system with real history, naive prompt strategies break down. (source: Dex Horthy (HumanLayer), "No Vibes Allowed: Solving Hard Problems in Complex Codebases")

The fix isn't a better prompt. The fix is treating context as a managed resource: one that gets designed, structured, and actively maintained across a session's entire lifecycle.

---

what goes into context (it's more than you think)

When engineers picture context, they picture the system prompt and the conversation. That's a fraction of what actually runs through the context window in a real agentic session.

Ankur Goyal at Braintrust analyzed token distribution across agentic prompts and found that tool definitions and tool responses dominate the token budget. The system prompt, where engineers spend most of their tuning effort, is often a small slice of total context. The bulk of what the model "sees" is tool output, prior conversation, and injected file content. (source: Ankur Goyal (Braintrust), "Five Hard Earned Lessons About Evals")

This has a direct implication. If you're only tuning your system prompt, you're optimizing the smallest slice. The 90% of context you're not managing (the raw tool responses, the uncompressed conversation history, the repeated injections of the same config file) is working against you.

One concrete example: shifting a tool's output format from JSON to YAML made a measurable difference in agentic task performance. The content was identical. The token footprint and parse characteristics changed. Tools are a communication channel with the model, not just an API wrapper, and they should be designed for what the model needs to see, not just mirrored from your existing APIs. (source: Ankur Goyal (Braintrust), "Five Hard Earned Lessons About Evals")

---

the WHAT/WHY/HOW structure for instruction files

There's a structural reason why most `CLAUDE.md` files fail to hold. They're written as flat lists: do this, don't do that, always use this component. That structure is easy to write and hard for a model to use selectively.

The context-as-cognitive-substrate framing offers a more durable lens: a spec is complete when it encodes not just the WHAT (the rule) but the WHY (the constraint behind it) and the HOW (the acceptable path when the rule meets an edge case). A model working from a rule list applies it by pattern-matching the current task against the rule text. A model working from a principle, one grounded in a stated reason, can generalize across situations the rule author didn't anticipate.

Debois makes the same observation from the build side: when you author context as reusable skills (structured instruction sets that encode intent and method, not just directives) those skills travel across projects, scale to teams, and survive model updates in a way flat instruction files do not. A skill that says "first figure out the package manager, then the ecosystem, then work through the steps with the user" encodes reasoning procedure, not just an outcome target. (source: Patrick Debois (Tessl), "Context Is the New Code")

The practical takeaway: every rule in your instruction file should have its reason next to it. The model can't recover gracefully from a rule it doesn't understand. The model can navigate edge cases on a principle it has internalized.

---

worked example: two versions of the same instruction

Consider one rule that should appear in every serious `CLAUDE.md`: run the test suite before declaring a task complete. Simple to state, routinely broken in practice. Not because the agent ignores it, but because of how it was authored.

The ad-hoc version, typical of a file that grew by accretion:

```
Always run tests before saying you're done.
Don't skip tests even if you think the change is small.
Run npm test. If it fails, fix it before finishing.
Never mark a task complete without running tests.
Tests must pass.
```

Five lines saying the same thing five ways. Each line was probably added after a violation. The model sees high token weight on “run tests” but no structure to reason from. At turn 30, under token pressure, with a change that feels trivial, the agent pattern-matches “small change, probably fine” and skips the step. The rule lost to live context noise.

The engineered version:

```
## Completion gate

**Rule:** Run the full test suite before any task is marked complete.

**Why this exists:** A change that passes casual inspection can still
break a downstream dependency. A failure caught inline costs far less
than one discovered after handoff.

**How to apply it:**
- Run `npm test` (or the project-specific equivalent from CONVENTIONS.md).
- If tests fail, diagnose and fix before proceeding. Do not suppress.
- Exception: if the test suite requires an external service unavailable
  in this session, document the gap in the task summary.

**Edge case:** documentation-only changes (no code changed) skip the
test run; note "docs-only, no test run" in the output.
```

Same rule. The engineered version is longer, but it costs less context budget as sessions grow, because the model can apply the specific clause that fits the current moment. “Is this docs-only? No. Is a runner available? Yes. Gate applies.” The decision tree is in the context; the model follows it rather than guessing.

Against the ad-hoc version, a session at turn 25 with a minor refactor frequently skips the test step. Against the engineered version, the same session applies the exception check, finds it doesn't apply, and runs tests. The WHAT/WHY/HOW structure isn't a style preference. It's the mechanism by which a rule survives a full session.

---

context as a lifecycle, not a file

If context is the new code, then context needs a development lifecycle the same way code does. The current state of most teams is the equivalent of writing software with no version control, no tests, and no deployment process. Just editing files and hoping.

Debois frames it as a five-stage loop: Generate → Test → Distribute → Observe → Adapt. (source: Patrick Debois (Tessl), "Context Is the New Code")

Each stage has real engineering work behind it. Generate means deciding what gets authored explicitly versus pulled dynamically from external sources: documentation, tickets, tool outputs. Test means running evals: not just "does the agent seem to be working" but structured assertions against defined criteria, repeated multiple times given non-determinism. Distribute means packaging context so it can be shared across a team and not degraded in translation. Observe means instrumenting agent behavior so you know when context gaps are causing failures, and feeding that signal back. Adapt means treating context files as living artifacts that get updated from observed failures, not static documents that accumulate rules until they collapse under their own weight.

Most teams are in the Generate stage only. They write instructions, deploy them, and wait for failures. The lifecycle view tells you that everything after Generate is where discipline compounds.

The Test stage is where most teams discover they've been guessing. Running evals against context isn't optional. It's the mechanism by which you distinguish "the agent seems to follow this rule" from "the agent follows this rule 87% of the time." LLM outputs are non-deterministic, which means a binary pass/fail check on a single run tells you almost nothing. Treat context quality the same way you treat system reliability: error budgets, pass rates across multiple runs, and regression tracking when you change anything. (source: Patrick Debois (Tessl), "Context Is the New Code")

---

diagnostic: where does your context practice stand?

Run this against your current setup. Each “no” is a gap worth closing.

1. **Do you know what percentage of your context budget is consumed by tool outputs versus instructions?** If you’ve never measured this, you’re managing blind.
2. **Does each rule in your instruction file include the reason behind the rule?** A rule without a WHY degrades to noise as session length grows.
3. **Have you ever run a deliberate eval against your instruction file?** Not a vibe check: a structured test that asserts specific behavior and measures pass rate across multiple runs.
4. **When an agent produces a bad output, do you update the context that produced it, or do you fix the output and move on?** Fixing the output is treating the symptom. Updating the context fixes the source.
5. **Do you have any instrumentation that surfaces which instruction failures recur across sessions?** Pattern detection on failures is the feedback loop that makes context improvement compounding rather than reactive.

A score of four or five out of five means your context practice is ahead of most teams. Two or fewer means you’re in prompt-only territory, and the four failure modes in the next chapter are already working against you.

If your instruction files are growing and your agent behavior isn’t improving in proportion, the problem isn’t the rules: it’s the structure. The next chapter names the four specific failure modes that turn well-intentioned context into soup.

Prompt Soup: The Four Failure Modes Killing Your AI Sessions

> "…it feels like trying to delegate tasks to an arrogant developer with severe dementia issues"
>
> source: user geosal, Cursor forum (source)

That's how a Cursor forum user described their AI coding agent. Not a prompt problem. Not a model problem. A context problem.

There's a name for what's happening. It's called **prompt soup**.

Prompt soup is a state of context window saturation where competing instructions, fragmented history, and raw tool outputs degrade the model's reasoning coherence. Think of the context window as the model's working RAM. When that RAM fills up with mechanical bloat (raw API responses, metadata, repetitive re-injected config files) the model doesn't crash. It degrades. Quietly. The output keeps coming; the reasoning stops holding together.

Four specific failure modes produce prompt soup. If you've used an AI coding agent for anything non-trivial, you've already hit at least two of them. This essay names them, shows you what they look like in the wild, and gives you a four-question diagnostic you can run right now.

Failure Mode 1: Context Rot

Context rot is the slow decay of session quality as the token budget fills. The model doesn't suddenly break. It drifts: early instructions lose weight, files it already processed become unfamiliar, decisions made in turn 3 get overwritten by turn 30.

Here's what it looks like from the inside:

> “You’re 40 minutes in, deep in a complex refactor, and the model starts forgetting things. It repeats itself. It loses track of what files it already edited. Then the session just dies. Context window full, conversation over, work lost.”
>
> source: dev.to/dibyanshu_kumar (source)

The session didn’t fail because the model got dumber. It failed because the context became incoherent. Early conversation turns, where the architect-level decisions live, are the first to be deprioritized as new tokens push them toward the edge of the window. The model literally cannot pay equal attention to everything in context; it weights recent tokens more heavily. So the vision you established at the start gets crowded out by the mechanical back-and-forth of later turns.

The sharp version of this failure has a name too:

> “It responded as if it had never seen that file before, showing no historical context”
>
> source: user geosal, Cursor forum (source)

That’s not a hallucination. The file was in the context. The model processed it. But by the time the question came up, so much other material had accumulated that the earlier processing didn’t survive as actionable signal. The context window is not a database with indexed retrieval. It’s a fixed-size sliding window of attention.

Context rot is the foundation failure. It makes every other problem worse.

Failure Mode 2: Instruction Drift

You spent an hour writing a `CLAUDE.md` (the persistent instruction file where you tell the agent how to behave in your project). You tuned it carefully. You told the agent which components to use, which patterns to follow, which shortcuts to never take. The agent read it at session start, confirmed it understood, and then, somewhere around tool call 15, stopped following it.

Instruction drift is what happens when behavioral instructions compete with the accumulated noise of a long session and lose.

> “These days Claude Code can barely even remember its CLAUDE.MD instructions.”
>
> source: user ApolloRising, Hacker News (source)

The reflex response is to make the instruction file bigger: add more rules, be more explicit, cover every edge case. But that makes the problem worse:

> “My CLAUDE.md has grown a lot because I was tired of Claude doing the same mistakes again and again and again… It’s now 44.7k chars… I tried reviewing manually the file but I can’t remove any parts of it risking Claude to do bad again.”
>
> source: GitHub Issue #2766 (source)

A 44.7k character instruction file is itself a soup ingredient. It consumes budget before the agent even starts working. And a dense, unstructured instruction block is hard for the model to selectively retrieve. It’s more likely to follow the instructions it most recently encountered, or the ones most syntactically similar to the current task, than the ones most semantically relevant.

The user’s description is accurate:

> “CLAUDE.MD is just a markdown file where I say ‘Hey Claude always use X agent for X component’… and then pray Claude remembers to use it.”
>
> source: user ApolloRising, Hacker News (source)

“Pray it remembers” is not an engineering strategy. But without a principled approach to how instructions are structured and when they’re injected, prayer is all you have.

Instruction drift is a retrieval problem wearing a compliance costume. The instructions aren’t being ignored because the model is disobedient. They’re being ignored because they’ve been buried.

Failure Mode 3: Token Bloat

Token bloat is the invisible budget drain. Unlike context rot (which you feel) and instruction drift (which you notice), token bloat is silent until you hit the wall.

The mechanism is simple: tool calls, MCP integrations (MCP, or Model Context Protocol, is the standard interface that connects external tools and data sources to an AI agent), and config injection each add tokens to the context. Those additions compound. Most users have no visibility into how fast their budget is draining or what's consuming it.

> "CLAUDE.md getting re-injected on every tool call. 50 calls × 10KB = 500KB gone."
>
> source: naqeebali-shamsi, Medium (source)

That's not a theoretical calculation. That's a working developer watching their context budget evaporate with no signal that it's happening. The same source documented a case that makes this concrete at scale:

> "Someone else's sandbox mode generated a 607K token system prompt. The context limit is 200K."
>
> source: naqeebali-shamsi, Medium (source)

A 607K token prompt for a 200K limit model. The session never had a chance. Every response was working against a context that had already exceeded capacity before the first user message arrived.

The downstream effect shows up as a dollar problem:

> "$200/month subscribers watching their 5-hour usage windows vanish in 90 minutes."
>
> source: naqeebali-shamsi, Medium (source)

Three hours of planned work, gone. Not because the agent failed, but because the context architecture consumed the budget before the work could happen. Token bloat is what happens when raw tool outputs are treated as prompt content: every MCP call that returns a 50KB JSON blob, every repeated file read, every un-summarized API response goes directly into the working memory with no compression, no filtering, no eviction.

The context window is a shared resource. Right now, most tool integrations treat it like it's infinite.

Failure Mode 4: Agent Loop Thrashing

The first three failure modes degrade silently. This one is visible, frustrating, and expensive.

Where context rot looks like silent drift and forgetting, thrashing looks like visible repetition: the agent loops through the same wrong answer out loud.

Agent loop thrashing is when an agent enters a cycle of failed attempts: trying an approach, hitting an error, backing up, trying again, hitting the same error from a different angle, and repeating. The agent isn't making progress. It's burning context tokens on approaches it already tried without recognizing the repetition.

> "Claude gets into a long cycle of 'wait that's not right.. hold on…'"
>
> source: user JamesSwift, Hacker News (source)

That ellipsis in the quote is doing real work. It captures the experience: the model catching itself mid-execution, correcting course, and then, because the context doesn't give it a clean map of what's already been tried, heading in the same wrong direction again.

One behavioral variant has its own name:

> "There's been a very significant increase in 'rush to completion' behavior."
>
> source: user koverstreet, Hacker News (source)

Rush-to-completion is thrashing in disguise. Instead of looping visibly, the agent shortcuts evaluation, skips verification steps, and ships a plausible-looking result that doesn't actually solve the problem. Both patterns have the same cause: the agent has lost track of what it already tried and what constraints ruled those approaches out.

This is a state management problem. A human developer who hits a dead end makes a mental note: "tried approach X, failed because Y." That note informs the next attempt. An agent working in a degraded context has no reliable way to maintain that map. The constraint log (the record of what failed and why) is buried somewhere in the conversation history, competing with thousands of other tokens for attention.

Loop thrashing is context rot plus no working memory plus no failure state tracking. It's what happens at the bottom of the soup.

what soup looks like

The four failure modes above are easier to recognize once you've seen them together in raw context. The following is a representative excerpt from a real agentic session: a `CLAUDE.md` that re-injects on every tool call, followed by back-to-back tool responses that land unsummarized, followed by fragmented conversation turns where the agent has lost the thread.

```
[SYSTEM: injected at turn 1]
Always use the Button component from @ui/core.
Never import directly from ./components.
Use TypeScript strict mode.
Run tests before marking done.
...
[430 more lines of project rules]

[SYSTEM: re-injected at turn 8 after tool call]
Always use the Button component from @ui/core.
Never import directly from ./components.
...
[same 430 lines again]

[TOOL: read_file, turn 8 response]
{"path":"src/api/orders.ts","content":"import { Request, Response }
from 'express';\nimport { db } from '../db';\nimport { OrderSchema }
from '../schemas/order';\n// ... 847 more lines of raw file content"}

[TOOL: search_codebase, turn 9 response]
{"matches":[{"file":"src/components/OrderForm.tsx","line":14,
"text":"import Button from './Button'"},{"file":"src/pages/checkout.tsx",
"line":22,"text":"import { Button } from '@ui/core'"},
... 43 more match objects with full file paths and surrounding lines]

[ASSISTANT: turn 10]
I see the issue. Let me check the import again.

[TOOL: read_file, turn 10 response]
{"path":"src/components/OrderForm.tsx","content": ... 612 lines}

[ASSISTANT: turn 11]
Wait, that's not right either. Let me look at the Button component
to understand the API surface.
```

Three failure modes are visible here. The `CLAUDE.md` re-injection at turn 8 is token bloat: the same 430-line instruction block consumed budget twice before the agent took a single meaningful action. The raw `read_file` and `search_codebase` responses are uncompressed tool output landing directly as

context: hundreds of lines of source code and match objects that the agent could have summarized to three relevant facts. The assistant turns at 10 and 11 are early loop thrashing: the agent is re-reading files it already processed, with no record of what the prior read established.

This context isn't broken. It's typical. Most agentic sessions accumulate this pattern within the first 15 turns.

The Four-Question Diagnostic

Before your next session, run this check. If you answer "yes" to two or more, you're building toward prompt soup.

1. Does your config file (CLAUDE.md or equivalent) exceed 5,000 characters? If yes: the file itself is a token bloat source, and long unstructured instruction blocks drift faster than short structured ones.

2. Do you regularly hit session limits mid-task on work you expected to complete in one pass? If yes: you have a token bloat or context rot problem. The budget is being consumed faster than your work requires.

3. Have you ever caught the agent doing something you explicitly told it not to do, but only after turn 20 or later in a session? If yes: instruction drift. Your instructions arrived but lost the retrieval competition.

4. Has an agent ever repeated an approach that already failed in the same session without acknowledging the prior attempt? If yes: loop thrashing. The failure state wasn't recorded in a way the agent could access.

Naming the failure modes only takes you so far. To stop generating prompt soup, you need a model of where context comes from and where it goes. That model has a name.

The Harness Model

You've called the API in a loop. You've added a retry on failure. You've strung tool calls together and called it an agent. And then you watched it degrade, thrash, forget, and die, over and over, in ways that felt unrelated to each other but weren't.

The problem wasn't the loop. The problem was that the loop had no runtime layer between "call the model" and "manage what the model sees." That gap is where the harness lives.

---

the context window is not a buffer

The dominant mental model, treating the context window as a static buffer you fill with text, is the root cause of most agentic failures. A buffer fills up. You deal with it when it's full. That's the wrong frame.

The context window is a **dynamic resource**: a fixed-capacity working memory that must be actively managed across every turn of a session. What goes in, when it goes in, in what form, for how long, and when it gets replaced: these are active engineering decisions, not passive accumulation.

The evidence for this is direct. Nominal token limits are often described as "vanity metrics": functional performance degrades 30–40% from context rot (the slow degradation of session quality as the token budget fills and early instructions lose attention weight) well before the physical limit is reached.

Models also exhibit what researchers call "context anxiety," a behavior where the agent prematurely wraps up complex tasks as it senses the context limit approaching. The degradation isn't binary. It's progressive. The agent doesn't crash; it quietly stops holding things together.

This means the context window demands the same engineering attention you'd give any constrained shared resource: careful allocation, active monitoring, and deliberate decisions about what stays and what goes.

An **agent harness** is the opinionated runtime layer that makes those decisions programmatically. It sits between your application logic and the model API. It decouples the reasoning engine (the model itself) from session state, so that what the model sees at each turn is a managed, purposeful snapshot rather than raw accumulated history.

---

harness versus loop: what actually differs

A loop calls the model. A harness manages what the model calls are allowed to see.

The distinction sounds small. In practice it's everything.

A naive agent loop looks like this: send message, get response, append both to conversation, repeat. Every turn, the conversation grows. Every tool call appends its output. The context fills. The agent degrades. There is no opportunity to intervene because there is no layer between the application and the raw context state.

A harness introduces that layer. It gives you handles on four things a loop doesn't:

Tool output management. Tool calls are the largest consumer of context budget in real agentic workflows, often larger than system prompt and user messages combined. (source: Ankur Goyal (Braintrust), "Five Hard Earned Lessons About Evals")

A harness can intercept tool responses before they hit the context, compress them, extract the signal, and inject a structured summary instead of raw output. An MCP (Model Context Protocol) call that returns 50KB of JSON doesn't need to land in the context as 50KB of JSON. The harness makes that decision before the model sees it.

Session state persistence. When a session ends or a context window fills, a loop loses everything. A harness can serialize agent state into a structured handoff (the decisions made, the constraints encountered, the current position in the task) that a fresh session can resume from without reconstructing from scratch.

The distinction between two approaches is clean: **compaction** uses LLM-generated summaries to compress history, but produces "shift-change amnesia": the nuance and decision logic that didn't survive the summary.

Compounding uses structured handoffs (YAML or JSON) to transfer state verifiably, so the next session's agent starts with the same effective context the previous one had.

Context-window paging. MemGPT pioneered the OS memory analogy: the context window is fast memory; external storage is slow memory. When active context grows beyond the working set needed for the current step, a harness can page data out to external storage and retrieve it on demand. The model works in a smaller, cleaner window; the full state persists outside of it.

Trajectory control. A context isn't just tokens; it's signal about what the session has been doing. A context full of repeated errors, corrections, and apologies trains the model's next-token predictions toward more of the same. The Dex Horthy formulation: if you yell at the agent repeatedly, the model learns that the next token should be wrong so you'll yell again. (source: Dex Horthy (HumanLayer), "No Vibes Allowed: Solving Hard Problems in Complex Codebases")

A harness can detect trajectory degradation and trigger a clean compaction, not because the window is full, but because the accumulated signal is working against the task.

---

the Manus pattern: stable prefixes and append-only context

One of the clearest demonstrations of harness discipline in production comes from Manus, where the central insight was that KV-cache (the key-value cache that lets the model reuse prior computation for stable context prefixes) hit rate is the single most important metric for production-grade agents. (source: Val Bercovici + Kellen Fox (WEKA), "Context Platform Engineering to Reduce Token Anxiety")

The KV cache reuses identical context prefixes to avoid reprocessing on every turn. A single token change at the start of a prompt can invalidate the entire cache, producing a 10× spike in latency and cost. (source: AgentSight (arXiv:2508.02736))

The Manus harness discipline follows directly: keep the system prompt and other stable elements at the start of the context as an immutable prefix. Append tool outputs and conversation turns after it, never before. When state

must change, mask the stale segment rather than remove it, because removal shifts the prefix and kills the cache. (source: AgentSight (arXiv:2508.02736); Kgent)

This is not a performance optimization bolted onto an existing architecture. It's a structural constraint that determines how the entire context gets built. The harness enforces it turn by turn.

The economic consequence is real. Cache misses cost the full input token price. Cache hits cost a fraction of that. In an agentic session with hundreds of turns, the difference between 40% and 90% cache hit rate can determine whether the session is economically viable at all. (source: Val Bercovici + Kellen Fox (WEKA), "Context Platform Engineering to Reduce Token Anxiety")

---

the five disciplines the harness orchestrates

A harness doesn't implement context management in one monolithic block. It orchestrates five distinct disciplines, each targeting a different failure mode. These disciplines get full treatment in Part II; this is the map.

Generate: deciding what goes into context in the first place. Not every piece of available information should be injected. The harness enforces selectivity: which files, which tool outputs, which history, at what level of detail, and when. Undisciplined generation produces token bloat before a single task has been attempted.

Retrieve: surfacing the right information at the right moment. Semantic retrieval engines (graph-RAG approaches being the most developed example) can reduce token waste by 65–70% by replacing broad file reads with targeted injections of semantically relevant fragments.

Instead of the agent running `grep` and `cat` across an entire codebase, the harness retrieves exactly the code paths relevant to the current step and injects only those.

Compress: reducing mechanical bloat without losing signal. Tool outputs, conversation history, and injected documents all accumulate. A harness applies intentional compaction proactively, at defined thresholds, not under pressure when the window is already full. Compaction at 40% window utilization produces better output than compaction at 95%. (source: Dex Horthy (HumanLayer), "No Vibes Allowed: Solving Hard Problems in Complex Codebases")

Route: directing work to the right context at the right scope. A single top-level agent context holding all state for a complex task degrades faster than three scoped sub-agent contexts, each responsible for a focused problem. Sub-agents are context-isolation mechanisms: each forks a fresh window, does focused work, and returns a compressed summary to the parent. (source: Dex Horthy (HumanLayer), "No Vibes Allowed: Solving Hard Problems in Complex Codebases")

Evict: clearing what's no longer needed before it becomes noise. Prior decisions, completed subtasks, and resolved constraints don't need to travel through the rest of the session. The harness evicts them on schedule, not when the window is already full. Eviction is proactive hygiene; compaction is emergency triage.

---

worked example: the harness applied to one task

Task: add a `/orders/{id}/cancel` endpoint to an Express service.

Discipline	What it does
Generate	Injects `CLAUDE.md` (the persistent instruction file for the project), the route file, and `OrderSchema`. Codebase and migration history stay out. Agent starts with ~8KB of purposeful context.
Retrieve	Agent needs the auth pattern from a sibling route. Harness fetches those 40 lines via semantic lookup; the rest of the file stays out.
Compress	Test runner returns 3,200 lines. Harness extracts the two failing test names and assertion messages (18 lines) and injects only those.
Route	Migration work forks a sub-agent (a scoped context for a focused subtask that returns a summary to the parent). Sub-agent returns: migration name, fields added, rollback path, result. Parent never sees the working conversation.
Evict	Once migration is confirmed, its four-line summary is evicted. Constraint resolved; carrying it forward adds noise.

At completion: original instructions, the final route file, an 18-line test result, a four-line migration summary.

---

avoiding the dead weight trap

There’s a real risk in harness design that emerges from early agentic system work: “dead weight infrastructure.” Harnesses are often built to compensate for specific model weaknesses. Context resets, hard-coded retry limits, and aggressive compaction strategies that made sense for a model with a 32K window become redundant overhead, and sometimes actively harmful, when the model supports a million tokens and handles long context more gracefully.

The discipline: build harnesses around durable problems (structured state transfer, KV-cache stability, tool output management) rather than around specific failure modes that may not survive the next model release. A lean State Transfer Protocol, verifiable, structured, and model-agnostic, compounds in value as models improve. A hard-coded compaction-at-80%-utilization rule becomes technical debt.

The signal to watch: “embedding lock-in,” over-optimizing chunking and retrieval for one model’s failure modes, creates an architecture that’s hard to migrate off when the model changes.

---

diagnostic: does your setup have a harness?

Answer these questions about your current agent architecture.

1. **When a tool call returns a large response, what happens to it before it hits the context?** If the answer is “it goes in as-is,” you have no compression discipline.
2. **When your context window approaches its limit mid-task, what is the recovery path?** If the answer is “start over,” you have no state persistence. If the answer is “run a summary and continue,” check whether that summary has lost the decision history you’ll need later.
3. **Do your sub-agents share a parent context, or does each one get a scoped context with only what it needs?** Shared parent context is not routing; it’s context bloat by another name.
4. **Is your system prompt at a stable prefix position that doesn’t shift as the session progresses?** If tool outputs or injected files can appear before the system prompt in later turns, your KV-cache utilization is degraded on every turn.
5. **Do you have a structured state format for session handoffs?** A text summary is compaction. A structured handoff (YAML or JSON with defined fields for current task state, constraints encountered, and completed steps) is compounding.

Three or more “no” answers means you’re calling the API in a loop with extra steps. The harness is the architectural intervention that makes the five disciplines implementable, and Part II walks through each one in detail.

The first discipline, Generate, is where context comes from and how it gets into the window. That’s where most of the avoidable waste happens, and it’s where the leverage is highest.

Generate: Engineering What the Agent Always Knows

> “CLAUDE.MD is just a markdown file where I say ‘Hey Claude always use X agent for X component’… and then pray Claude remembers to use it.”
>
> source: user ApolloRising, Hacker News (source)

That quote is not a complaint about a bad prompt. It’s a description of a broken engineering practice. The developer did the work. They wrote the rules. The agent acknowledged them. And then the rules became noise somewhere between session start and tool call 20.

Generate is the first discipline of context engineering: deciding what goes into the always-loaded context, how it’s structured, and how to verify the agent actually uses it, not just at turn one, but at turn 40.

---

what always-loaded context actually is

Every agent session starts with a system prompt. In Claude Code, it’s the built-in model instructions plus your `CLAUDE.md` (the project-level instruction file where you define the agent’s role, rules, and constraints). In a custom agent, it’s whatever you push into the system message before the first user turn. Either way, this block is the context the agent is guaranteed to see at session start. It’s the one piece of context you fully control.

Patrick Debois frames the significance precisely: reusable instructions are being standardized as files like `CLAUDE.md`, a set of instructions that describe how to solve a problem, transforming what used to be explicit code into context that drives agent behavior (source: Patrick Debois (Tessl), “Context Is the New Code”). He draws an analogy to a DevOps shift: in 2009 people asked “what if ops looked more like dev?” The current shift is “what if context is the code?” If context is the new code, then your `CLAUDE.md` is your codebase. Most `CLAUDE.md` files are written the way codebases looked before version control: no structure, no tests, no modularity, and a growing mass of accumulated corrections.

The always-loaded context has a specific job. It answers three questions the agent needs at every turn, not only at session start:

- **WHAT** the agent is. Its role, its scope, what “done well” means in this context.
- **WHY** the constraints exist. The reasoning behind rules, not the rules alone.
- **HOW** to behave in the recurring situations this agent faces every session.

Agents that only receive the WHAT and HOW, rules without rationale, drift faster when they encounter situations the rules didn’t anticipate. The model falls back on training priors. If your `CLAUDE.md` says “always use the auth middleware” but doesn’t say why, the model treats apparent exceptions as actual exceptions. The WHY is what enables a constraint to survive novel situations. A rule without a reason is a rule that has an invisible off-switch.

Consider the difference between these two instructions:

```
Bad:  Never modify user_permissions directly.

Good: Never modify user_permissions directly — use PermissionsService.
      We had a privilege escalation incident in March 2025 because
      three separate code paths were mutating this table independently.
      PermissionsService enforces the audit trail.
```

The second version survives edge cases the first doesn’t. When the agent encounters a shortcut that looks safe, “we had a privilege escalation incident” is a reason to stay cautious. “Never do X” is a reason to wonder if this situation is an exception.

---

the anatomy of a CLAUDE.md that works

A `CLAUDE.md` that doesn’t drift has a deliberate structure. Not a list of rules. An architecture.

Identity and scope. Open with one tight paragraph. What is this agent for? What’s the bounded domain of decisions it should make without asking? What’s outside its authority? This anchors every subsequent decision the agent makes when the instructions don’t directly cover a case.

Architectural invariants. These are the rules that must never be violated. Five to ten, no more. Each gets one sentence stating the constraint, one sentence of rationale, and, where applicable, a pointer to where the history lives. Not preferences, not defaults: actual invariants. The distinction matters because the model treats them differently. An invariant named as such carries weight in ambiguous situations that a “best practice” doesn’t.

Recurring patterns. The positive patterns the agent should follow consistently. How to name things. Which utilities to use for which problem classes. How errors should surface. How tests should be structured. Concrete examples beat abstract descriptions here by a wide margin. A common pattern in agentic-coding setups demonstrates this: effective no-deferring rules don’t say “don’t ask if you should continue.” They explain that momentum is the scarce resource, that checkpoint questions interrupt flow without adding value, and they name the specific acceptable alternative: finish a step and immediately start the next. The reasoning survives situations the rule’s author never anticipated. Rules written as bare imperatives don’t.

Anti-patterns with context. The things the agent must not do. These are often the most valuable section because they encode hard-won experience: the approaches that look right and aren’t, the shortcuts that work locally and fail at scale, the patterns that worked six months ago and were replaced by something better. Without this section, agents rediscover the same dead ends by independent reasoning and reach the same wrong conclusions.

Verification triggers. Instructions that close the compliance loop within the context itself. “After any database schema change, run `npm run db:validate` before proceeding.” “Before writing any API endpoint, state which authentication pattern you’re using and confirm it complies with the auth invariant above.” These don’t just tell the agent what to do; they force the constraint back into the active context at the moment it matters.

---

why CLAUDE.md bloats and what that costs

The bloat trajectory is consistent across teams that have been running agents for more than three months. The agent does something wrong. A rule is added to prevent it. The agent does something else wrong. Another rule is added. The file grows with each correction, and after a while the developer can’t

safely remove anything because they've lost track of which rules are still preventing which failures. One GitHub issue documented the endpoint of this trajectory exactly:

> "My CLAUDE.md has grown a lot because I was tired of Claude doing the same mistakes again and again and again… It's now 44.7k chars… I tried reviewing manually the file but I can't remove any parts of it risking Claude to do bad again."
>
> source: GitHub Issue #2766 (source)

The rules aren't wrong. The structure is. A 44,700-character instruction block has properties that actively work against compliance, independent of how carefully each rule is written.

Token cost. That block consumes roughly 11,000 tokens before the first user message. In a 200K context window, that's 5.5% of the budget consumed before any work starts. In a session that runs 100 tool calls with substantial outputs, a bloated `CLAUDE.md` is the most expensive line item that produces the least incremental work.

Retrieval competition. The model doesn't parse `CLAUDE.md` the way a human reads a document top-to-bottom. It computes attention across all tokens in context simultaneously. A rule buried at character 30,000 of a dense, unstructured block competes against every other rule, the user's current request, and all accumulated tool outputs. It doesn't matter how important the rule is if it can't win that competition at the moment it's relevant. Jared Zoneraich observes that Claude Code warns in yellow when context exceeds 40K characters specifically because dense system prompts start degrading instruction-following at that scale (source: Jared Zoneraich (PromptLayer), "How Claude Code Works").

Semantic dilution. A 44K character block is nearly impossible to read as a human, which means it's nearly impossible to audit. Over time, rules contradict each other. Later rules quietly supersede earlier ones without explicitly doing so. The document becomes a palimpsest: layers of corrections written over earlier corrections, with the original intentions partially legible but not fully recoverable.

Maintenance paralysis. The GitHub commenter above couldn't remove anything because the coupling between rules and prevented failures had become implicit. They'd lost the map. This is the same problem as technical debt in codebases: when you can't safely refactor, you can only accumulate.

The fix is not to write fewer rules. It's to structure the document so the most important rules win the retrieval competition reliably, and to split task-specific instructions out of the global `CLAUDE.md` into scoped files that load only when the relevant task begins. That split is the boundary between Generate and Retrieve.

---

the verification trap

The most dangerous failure mode in the Generate discipline isn't a badly written `CLAUDE.md`. It's a well-written one that you've stopped verifying.

The trap works like this: you invest real time in the instruction file. You structure it. You add rationale. The agent reads it at session start, and for the first few turns it behaves exactly as specified. You ship code. A week later, without noticing the change, the agent stopped following a rule somewhere around turn 20 of a long session. The rule wasn't ambiguous. The rule wasn't wrong. The rule got buried under accumulated tool outputs, conversation history, and retrieved content, and the model's behavior quietly reverted to training defaults for that class of decision.

This is instruction drift. It's invisible unless you build for it. The agent won't tell you it stopped following your architectural rules. It'll stop following them.

Two patterns reliably close this gap. The first is assertion-based verification: structure your `CLAUDE.md` to require explicit confirmation at key decision points. "Before writing any API endpoint, state which authentication pattern you're using and confirm it complies with the auth invariant above." This isn't bureaucracy; it's forcing the constraint back into the active context at the moment of application. The agent can't comply silently; it has to articulate the constraint, which means the constraint is actively in attention when the decision is made.

The second is eval-based verification at the `CLAUDE.md` level. Patrick Debois describes the full context lifecycle as Generate → Test → Distribute → Observe → Adapt (source: Patrick Debois (Tessl), "Context Is the New Code"). Most teams only do Generate. The Test step (running systematic evals against your instruction set to verify compliance) is what separates context that works from context you hope works. When you change two lines in your `CLAUDE.md`, you don't know the full downstream impact unless you test it. A simple eval

harness: 20 test prompts that each exercise a specific rule, with pass/fail criteria defined. Run it every time you change the file. Track compliance over time.

Evals for context aren't optional. They're how you close the loop between instruction authoring and verified behavior. Without them, you're operating on anecdote: "it seemed to follow the rule last Tuesday." With them, you have a signal. The eval pass rate on a given rule tells you whether the rule is effective, whether it's being respected under load, and whether it survived the last `CLAUDE.md` change intact.

---

scoping: what belongs in CLAUDE.md versus what doesn't

The `CLAUDE.md` is the always-loaded context. "Always-loaded" has a cost implication: every token in that file is paid on every session call. Information that's only relevant to specific task types shouldn't be in a document that loads regardless of task type.

The scoping question is: what does the agent need to know on every task, without exception? That's the only content that belongs in `CLAUDE.md`. Everything else belongs in task-scoped context files that load conditionally.

Concrete examples of what belongs in the global file: - The agent's role and authority boundaries. - Invariants that apply across all code in this project (auth patterns, error handling, security constraints). - Anti-patterns with enough incident history that any task could encounter them. - Verification triggers that apply to any file modification.

Concrete examples of what doesn't: - Documentation patterns for the API layer specifically, only needed when working on API code. - Database migration procedures, only needed during schema changes. - Test structure conventions, only needed when writing tests. - Library-specific patterns for dependencies not all code touches.

The second category should live in scoped instruction files that the agent loads through the Retrieve discipline when the task type warrants them. This pattern keeps the `CLAUDE.md` small enough to win the retrieval competition on every rule it contains, while keeping the full instruction set available on demand.

Patrick Debois notes that reusable context pieces are evolving toward a library format: chunks of context that can be packaged, versioned, and loaded when needed (source: Patrick Debois (Tessl), "Context Is the New Code"). Your scoped instruction files are an early implementation of this model. Each file is a context package for a specific class of task. `CLAUDE.md` is the core package that's always installed. Task-specific packages load when the task matches.

---

testing your CLAUDE.md like it's code

If context is the new code, then `CLAUDE.md` changes need what code changes need: tests that verify behavior before and after the change, and a regression baseline that confirms you didn't break something you weren't trying to fix.

Most teams skip this. They edit the file, observe the next few sessions anecdotally, and declare success if nothing obvious goes wrong. The problem is that instruction drift is a slow failure; it shows up at turn 30 of a long session, in edge cases that don't appear in the first few tasks after a change. Anecdotal observation doesn't catch it.

A context eval harness is a set of test prompts (usually 15 to 30) where each prompt exercises a specific rule or invariant, and you've defined pass/fail criteria in advance. "Given this system state and this user request, the agent should use the PermissionsService, not the raw database call." Run the harness after every `CLAUDE.md` change. Track pass rates over time. When a rule's pass rate drops below a threshold, the rule is either broken or the instruction for it is no longer effective.

Debois describes this as a hierarchy from linting to end-to-end evals: format validation first, then LLM-as-judge scoring for individual rules, then full task execution with behavior verification (source: Patrick Debois (Tessl), "Context Is the New Code"). You don't need all three levels from day one. Start with: for each rule in your `CLAUDE.md`, write one test prompt where the rule should clearly apply. Confirm the agent applies it. That baseline is more than most teams have, and it catches regressions that otherwise take days of production observation to surface.

The non-determinism of LLM outputs means no single test run is conclusive. Run each test prompt three to five times. Measure the pass rate. A rule with a 60% pass rate is a rule with an off-switch; it applies when the stars align and fails when context is noisier. Target 90%+ for invariants. For preferences and defaults, 75% is acceptable if the failures are graceful.

The distribution of failures tells you where to invest: failures concentrated in long sessions indicate instruction drift (the rule is losing the retrieval competition as context grows). Failures concentrated on specific task types indicate scoping issues (the rule's phrasing doesn't generalize to that task class). Failures distributed randomly indicate an underpowered rule (the reasoning isn't strong enough to anchor the constraint against model priors).

---

worked example: a CLAUDE.md, refactored

Take a real `CLAUDE.md` from a mid-size TypeScript project. Before refactoring, it looks like this:

```
# Rules
- Always use the Button component from @ui/core
- Never import directly from ./components
- Use TypeScript strict mode
- Run tests before marking done
- Never modify user_permissions directly
- Use PermissionsService for all permission changes
- Use pnpm not npm
- Always add error boundaries around async components
- Don't use any
- Check for existing utilities before writing new ones
- After any schema change run db:validate
- Use React Query for all data fetching
- Never call the database from a component
- Log errors to Sentry not console.error
...
[31 more rules, no rationale, no structure]
```

That's the file that produces the 44,700-character trajectory. Every rule is correct. None of them will survive turn 30 of a long session because there's no hierarchy, no rationale, and no way for the model to distinguish an invariant from a preference.

Here is the same file, refactored into WHAT / WHY / HOW:

```
## WHAT this agent is

You are the TypeScript frontend agent for the Orders platform.
Your authority: UI components, API integration, client-side state.
Outside your authority: database schema, backend services, auth logic.
When a task touches backend code, stop and flag it rather than proceed.

## WHY the constraints exist

PermissionsService constraint: we had a privilege escalation incident
in March 2025 because three code paths mutated user_permissions
independently. PermissionsService enforces the audit trail.
Evidence: incident-reports/2025-03-privilege-escalation.md

Direct component imports constraint: ./components is legacy. It was
replaced by @ui/core in Q4 2024. Direct imports create divergence
that breaks the design token pipeline. This was a 3-sprint cleanup.
Don't reopen it.

Database-from-component constraint: mixing data fetching and rendering
makes components untestable and breaks our SSR boundary. All data
fetching goes through React Query hooks.

## HOW to handle recurring situations

**Adding a new UI component:**
1. Check @ui/core exports first. Use what exists.
2. If a new component is needed, place it in packages/ui-core.
3. Never add components to ./components.

**Any database schema change:**
After completing the change, run `pnpm run db:validate` before
marking done. State the result explicitly.

**Any API endpoint:**
Before writing, state which authentication pattern you're using
and confirm it matches the auth invariant in the WHY section.

**Error handling:**
Route to Sentry. Never console.error in production code.
Pattern: `logger.error(err, { context: 'component-name' })`.
```

The annotations:

The `## WHAT` block (lines 1-6) defines role and authority. One paragraph. It gives the model a first-principles basis for decisions the rules don’t explicitly cover.

The `## WHY` block (lines 8-21) carries the rationale for the three highest-stakes constraints. Each one names the incident or the cost. When the model encounters an apparent exception (“this case seems different”), the WHY is what it has to check that intuition against.

The `## HOW` block (lines 23-36) covers the recurring task types this agent faces every session. These are not rules; they are procedures. Each one is a short checklist the model can follow without re-deriving the correct behavior from first principles.

The full refactored file is roughly 350 words instead of 44,700 characters. Every rule that was task-specific (migration procedures, test structure, library-specific patterns) has been moved to scoped files that load via Retrieve when the task type matches. The always-loaded context now contains only what the agent needs on every task without exception.

---

diagnostic: is your always-loaded context working?

Run this check before your next session. Score one point for each “yes.”

1. Does your `CLAUDE.md` exceed 8,000 characters? If yes: you’re consuming more than 2,000 tokens before work starts, and retrieval competition for rules buried late in the document is significant. Restructure or split.

2. Does any rule appear without a rationale (no “because,” no “after incident X,” no “reason:”)? If yes: that rule will drift when the agent encounters an apparent exception. A constraint without reasoning has no defense against edge cases.

3. Have you added more than three rules in the last 30 days without removing any? If yes: you’re in the bloat trajectory. Audit the oldest rules against actual failures. Rules that aren’t preventing observed failures are overhead.

4. Can you name, right now, the five most important rules in your `CLAUDE.md` in order of priority? If no: your document has no hierarchy. The model has no guidance for which rules win when two conflict, and they will conflict.

5. Does your `CLAUDE.md` contain instructions that only apply to a specific task type (API endpoints, migrations, tests)? If yes: those instructions are loading on every session regardless of task type. Move them to scoped files.

6. Have you verified that the agent follows a specific rule at turn 25 of a long session, not just at session start? If no: you've been measuring compliance at load time. Instruction drift is invisible unless you look for it at use time.

7. Do you have evals, even informal ones, that test rule compliance against a defined set of test prompts? If no: you're in "pray it remembers" territory. That's not an engineering strategy; it's a hope that degrades with session length and context window saturation.

Three or more "yes" answers indicates a Generate problem. The always-loaded context is overcrowded, under-structured, unverified, or some combination of all three. The work of fixing it starts here, with the structure, before adding more rules.

A well-engineered `CLAUDE.md` establishes the context the agent carries everywhere. But always-loaded context can only cover what you can anticipate in advance, at the cost of pre-loading it regardless of relevance. The second discipline handles everything else: the information that's too large to carry everywhere, too specific to pre-load, and only needed at the exact moment a particular task begins.

Retrieve: Pulling the Right Context at the Right Moment

> “The agent needed to search across hundreds of spans. Before, all that data lived in the main conversation. After, the main agent delegates the search to a sub-agent that handles the heavy lifting. The main conversation can stay small.”
>
> source: Sally-Ann Delucia, Head of Product at Arize (source: Sally-Ann Delucia (Arise), “Hierarchical Memory: Context Management in Agents”)

That architecture shift, from dumping everything into context to loading only what’s needed when it’s needed, is the Retrieve discipline in practice.

Retrieve is the second discipline of context engineering: designing the systems that surface just-in-time context on demand, without saturating the context window with information the current task doesn’t require.

If Generate answers “what does the agent always carry,” Retrieve answers “what does the agent reach for.” The answer changes with every task. The mechanism that produces the right answer, reliably and cheaply, is what this chapter builds.

---

why retrieval fails before you fix it

Most teams discover they need retrieval when their `CLAUDE.md` (the project-level file that holds the agent’s persistent instructions) stops scaling. The instruction file grows too large. They move task-specific content into separate documents. They build a rough search mechanism on top of those documents. They call it RAG. Then the agent starts missing things it should know, citing things that aren’t true, and consuming context budget at a rate that doesn’t match the volume of work being done.

The naive RAG failure is consistent across implementations. Peter Werry’s taxonomy, built from shipping the Unblocked context engine at production scale, identifies three specific mechanisms (source: Peter Werry (Unblocked), “Mergeable by default: Building the context engine to save time and tokens”):

Satisfaction of search. The agent finds one plausible chunk, stops looking, and proceeds with incomplete context. The term comes from radiology: a technician spots one abnormality and misses a second, more dangerous one. In retrieval, the "second abnormality" is the architectural context, the historical rejection, the reason a particular approach was tried and abandoned. That context lives in pull requests, incident reports, and decision records, not in the document the vector search surfaced first. The agent's retrieval confirmed it found something relevant. It didn't confirm it found everything relevant.

Lack of personalization. Raw vector search over a large knowledge base returns chunks that are topically relevant in the abstract, not to the specific codebase, team, and constraints the current task lives in. A query about authentication patterns in a polyglot organization might surface five valid approaches from five different services, none of which reflects the decisions this team made for this component. The agent picks from the menu without knowing which item the kitchen actually serves.

Inability to resolve conflicts. Naive retrieval can't judge recency, authority, or truth. If the main branch and a Slack conversation contradict each other about how a component should behave, the retrieval system surfaces both and delegates reconciliation to the agent. That reconciliation is non-deterministic and usually wrong. The agent has no basis for knowing which source is authoritative; it optimizes for plausibility and produces a result that satisfies neither constraint.

Research on implementing constraint layers for autonomous agents names the underlying dynamic: natural language instructions and expanded context windows are insufficient for maintaining systemic integrity because of "signal dilution": the correct constraint is present in the retrieved context, but competing chunks dilute its weight to the point where it doesn't govern the decision (source: Drift; COBALT-TLA). You cannot solve a signal problem by adding more signal. The architecture of the retrieval system has to change.

The common non-solution is a bigger context window. Peter Werry addresses this myth directly: even context windows of one million tokens are insufficient for most organizations' full knowledge base. More critically, scaling window size doesn't solve reasoning across disparate data sources, can't determine what's true versus false, and at projected sizes of 10M to 50M tokens, the cost and latency become operationally unacceptable (source: Peter Werry

(Unblocked), “Mergeable by default: Building the context engine to save time and tokens”). Bigger windows are a capacity solution to a comprehension problem. They don’t work.

---

semantic chunking: the foundation naive RAG skips

Character-count chunking is the original sin of retrieval pipelines. Split a document every 512 characters, embed the chunks, index them, call it done. The problem is that 512 characters almost never maps to a semantic unit. A chunk might contain the conclusion of one section and the beginning of the next. The embedding represents a mixture of both. Retrieval surfaces the chunk because it partially matches the query; the context the agent receives is coherent to a similarity scorer but incoherent as knowledge.

Semantic chunking changes the split criterion from character count to meaning boundary. Chunk at paragraph breaks. Chunk at section headers. Chunk at logical transitions, where a code comment shifts from describing what to describing why. The chunk should represent exactly one coherent thought, because when you embed it, you’re embedding a unit of meaning, not a span of text. That distinction is what makes retrieval precision possible.

Two specific patterns improve chunking quality beyond boundary selection alone.

Parent-child chunking. The child chunk is the small, semantically tight unit used for embedding and retrieval. The parent chunk is a larger section (several paragraphs, a full function, a complete decision record) that contains the child. When retrieval surfaces a child chunk, the system also injects the parent. This gives the agent precision in matching (from the child embedding) and context for interpretation (from the parent). The child chunk might be a single sentence stating a constraint; the parent is the full explanation of why the constraint exists and what it prevents.

Contextual embedding. Before embedding a chunk, prepend it with a short summary of its containing document. “This is an architectural decision record from March 2025 about the authentication service refactor. The following section covers the rejected approach:” followed by the chunk content. The embedding now captures both the local semantic content and its document-level context. Retrieval becomes more precise because the vector represents the chunk in context, not the chunk in isolation.

Metadata matters as much as the chunk content. A chunk without metadata is a decontextualized text fragment. Attach, at minimum:

- **Source path and document type.** An architectural decision record carries different authority than a code comment, which carries different authority than a Slack message.
- **Timestamp.** Recent sources should win ties in conflict resolution.
- **Author or team.** Expert weight changes the retrieval calculus. Feedback from a recognized domain expert on a pull request carries more authority than a comment from a contributor new to the codebase (source: Peter Werry (Unblocked), "Mergeable by default: Building the context engine to save time and tokens").
- **Status.** Active, deprecated, superseded. Retrieving a deprecated approach without flagging its status and what superseded it is actively harmful.

The metadata is what enables anything more sophisticated than cosine similarity scoring. Without it, you have embeddings. With it, you have addressable, filterable knowledge.

---

hybrid search: why cosine similarity alone is insufficient

Pure vector search retrieves by semantic similarity. It finds chunks about the same topic as the query. It handles "what does our codebase do for session management" well. It handles "what is the exact value of `MAX_SESSION_TIMEOUT_SECONDS`" poorly, because exact-match queries have no semantic similarity to anchor. The query and the answer look different in embedding space even though one directly answers the other.

This is the category of query that breaks vector-only retrieval silently. The retrieval system returns something, it always returns something, but it returns an approximate semantic neighbor, not the precise answer. The agent proceeds with the approximate information and produces a confident wrong result.

Keyword search (BM25 or equivalent) handles exact-match queries natively. Proper nouns, identifiers, specific values, version numbers, configuration keys: these are precisely the cases where keyword precision matters and vector similarity fails. Keyword search also handles cases where the

vocabulary of the query closely matches the vocabulary of the answer: documentation written with consistent terminology retrieves cleanly via keyword search even when the semantic embedding is noisy.

Hybrid search combines both. Run the query through vector search for semantic coverage, run it through keyword search for lexical precision, merge and deduplicate the result sets. Standard fusion approaches (reciprocal rank fusion is common) produce a merged ranking that captures both signal types. The weights are tunable based on query type: boost semantic weight for conceptual questions, boost keyword weight for identifier lookups. More sophisticated systems classify the query type and route it accordingly.

Knowledge graphs add a third retrieval mode that both vector and keyword search miss: multi-hop relational queries. Stephen Chin's work on graph RAG identifies the specific failure case (source: Stephen Chin (Neo4j), "Context Engineering: Connecting the Dots with Graphs"). "Which components depend on the auth service, and which of those components have open vulnerability reports filed in the last 90 days?" This query requires traversing a relationship graph. Vector search returns chunks about the auth service. Keyword search returns documents mentioning vulnerabilities. Neither surfaces the connected path through the graph.

Graph retrieval adds nodes (components, people, events, decisions, files) and edges (depends-on, authored-by, related-to, supersedes, was-fixed-by). A query that requires following edges becomes answerable. The agentic graph traversal variant described in Chin's demo works iteratively: the agent retrieves the graph schema first, then fires targeted Cypher queries, following relationships step by step until the answer is fully assembled. This is slower than two-pass vector search. For multi-fact questions that require traversing provenance, ownership, and dependency chains, it's the right architecture (source: Stephen Chin (Neo4j), "Context Engineering: Connecting the Dots with Graphs").

The practical retrieval stack for most production agent systems:

```
query
  → query classifier (conceptual? exact-match? relational?)
  → hybrid search: vector (semantic) + BM25 (lexical)
  → optional: graph traversal for relational queries
  → merged candidate set (50–100 chunks)
  → reranker (see next section)
  → metadata filter (recency, authority, access control, status)
  → final 3–5 chunks injected into context
```

Each layer eliminates a category of failure that the preceding layers can't prevent on their own.

---

cross-encoder rerankers: the second-pass precision filter

Retrieval pipelines have two distinct jobs. The first is recall: surface every chunk that could be relevant. The second is precision: put the most relevant chunks first so that what actually lands in the agent's context is the best available signal, not a chunk from the right neighborhood.

Vector search and keyword search both optimize for recall at scale. They're fast and have reasonable precision. "Reasonable" is insufficient when you're handing the top three chunks to an agent that treats them as ground truth.

Cross-encoder rerankers handle precision. After the initial retrieval (hybrid, vector, or keyword) pass the original query and each candidate chunk together through a cross-encoder model. Unlike the bi-encoder architecture used for vector search (which embeds query and chunk separately and then compares them), the cross-encoder sees both simultaneously as a single input and scores their relevance jointly. Joint encoding captures semantic interactions that separate embeddings miss: a query that uses different vocabulary than the chunk but means the same thing; a chunk that contains the answer but not the query terms; a multi-sentence chunk where only the third sentence is relevant.

The cost of cross-encoding is significant per pair, roughly 10 to 20 times the cost of a vector similarity lookup. This is why you use it as a second pass over a small candidate set (20 to 50 chunks from the initial retrieval), not over the full index. The two-stage architecture gets you the scale efficiency of approximate search plus the precision quality of exact cross-encoding. You pay for precision only where it matters: the final ranking.

After reranking, a metadata filter applies the constraints that scoring can't capture. A highly-relevant chunk from a deprecated branch should lose to a slightly-less-relevant chunk from main. A private channel's context shouldn't surface for users without access permissions. A chunk tagged "superseded" should be flagged even when retrieved, or excluded. Scoring tells you what's most relevant. Metadata tells you what's allowable. Both constraints apply.

---

the iceberg problem: what retrieval can't find

Retrieval only finds what's been written down and indexed. This sounds obvious. Its implications are less obvious.

Peter Werry describes what he calls the iceberg problem: the code that compiles is visible above the waterline; the reasons behind the code (why an approach was rejected, what incident drove a constraint, what the team tried in 2024 and abandoned) are invisible below it (source: Peter Werry (Unblocked), "Mergeable by default: Building the context engine to save time and tokens"). An agent equipped with excellent retrieval infrastructure still can't surface knowledge that was never committed to text. The architectural decision made in a whiteboard session that was never written up. The constraint that lives only in a senior engineer's memory. The incident whose lesson was applied to code but never documented.

This gap can't be closed by retrieval tooling. It requires a knowledge capture practice: the habit of writing decision records, annotating rejected approaches, committing post-incident learnings as documents alongside the code changes they motivated. Retrieval amplifies what's in the knowledge base. It doesn't fill the knowledge base.

Three other cases where retrieval is the wrong tool:

The information doesn't exist yet. A new library, a new API, a component that predates your documentation effort. Retrieval returns nothing useful or returns adjacent content that misdirects. The right tool here is an explicit fetch into context: download the documentation, add it directly, not a search against a knowledge base that doesn't contain it.

The chunking is wrong. The answer to the agent's implicit question is split across two chunks that share no surface similarity. The query finds chunk A and misses chunk B. Until you fix the chunking, retrieval machinery improvements won't help. The knowledge unit has to be whole before retrieval can surface it whole.

The agent doesn't know what to ask. Retrieval is pull-based. The agent formulates a query. If the agent doesn't know to look for a constraint, retrieval silently returns irrelevant results. There's no error. There's no signal that something was missed. This failure mode is invisible without deliberate instrumentation: logging what was retrieved, what the agent used, and what was missed in tasks that produced wrong answers.

---

when retrieval is working and when it isn't

The diagnostic signals for retrieval failure are distinct from the diagnostic signals for Generate failure. Generate problems manifest as rules being ignored. Retrieval problems manifest differently depending on the mechanism.

Retrieval returning irrelevant content is a chunking or embedding problem. The semantic units aren't coherent, and similarity search is matching noise.

Retrieval returning nothing relevant is a knowledge gap. The information isn't in the index. Fixing the retrieval pipeline won't help.

Retrieval returning the right type of content but consistently missing the most important piece is a reranking or metadata problem. The right chunk exists; it's losing the precision competition.

Agent making errors despite retrieval surfacing correct content is a different problem entirely: the injected context isn't winning the attention competition in the full context window. This is often a compound symptom: too much other material accumulated, the retrieved chunk is too short to anchor the decision, or the chunk lacks the parent context that makes the constraint legible.

Distinguish these before adding retrieval complexity. A team that builds a sophisticated hybrid search and reranking pipeline when the real problem is missing documentation is building infrastructure for a knowledge gap. The gap doesn't get smaller because the search got faster.

---

worked example: retrieval applied to a single query

Scenario: the agent is about to write a new API endpoint and needs to know how the auth middleware was changed last week.

Query issued: `"auth middleware changes last week"`

Initial hybrid search returns five candidates:

```
Chunk A  [score: 0.91, type: pull-request, date: 2026-05-19]
"PR #1842: replace session-cookie auth with JWT bearer tokens.
Auth middleware now validates RS256 signatures. All routes
that used req.session.userId now use req.auth.sub. Migration
guide in docs/auth-migration.md."

Chunk B  [score: 0.87, type: code-comment, date: 2026-05-19]
"// TODO: remove legacy session fallback after 2026-06-01
// added in PR #1842 as temporary backward compat shim"

Chunk C  [score: 0.84, type: slack-message, date: 2026-05-17]
"heads up: auth middleware PR landing next week, will break
anything reading req.session.userId directly"

Chunk D  [score: 0.79, type: code-comment, date: 2024-11-03]
"// auth middleware: validates API key from Authorization header
// added for service-to-service calls, not user sessions"

Chunk E  [score: 0.76, type: documentation, date: 2026-01-15]
"Authentication overview: the platform supports three auth
patterns: session cookie, JWT bearer, and API key..."
```

What the reranker does:

The cross-encoder sees the query and each chunk as a pair. Chunk A ranks first (direct answer: the change, the date, the new field name). Chunk B ranks second (adds the deprecation timeline, directly relevant to the agent's task). Chunk C drops to fourth because it's a prediction, not a record of what happened. Chunk D drops to fifth: it's about API key auth, not the JWT migration. Chunk E stays low: it's an overview from January that predates the change.

What gets injected:

Chunks A and B only. The metadata filter also confirms both are status: active and source: main-branch.

What the agent sees:

The JWT migration fact, the new field name (`req.auth.sub` not `req.session.userId`), and the deprecation deadline. Three actionable facts in roughly 80 tokens. Not 400 tokens of candidates the reranker would have deprioritized.

What was lost, and why that's fine:

Chunks C, D, and E are excluded. Chunk C is redundant (A is more authoritative). Chunk D is a different auth path. Chunk E predates the change. The agent has everything it needs to write the endpoint correctly and nothing that would misdirect it.

The pipeline: hybrid search for recall, cross-encoder reranker for precision, metadata filter for authority. Each layer eliminates a category of error the prior layer can't prevent.

---

diagnostic: is your retrieval working?

Run this against your current retrieval setup. Score one point per "yes."

1. Are your chunks split on semantic boundaries (paragraph breaks, section headers, logical transitions) rather than character count? If no: your embeddings represent text fragments, not concepts. Precision will be low regardless of everything built on top.

2. Does every chunk have attached metadata including source path, document type, timestamp, and author or team? If no: conflict resolution defaults to scoring alone. Recency and authority are invisible to your system.

3. Are you running keyword search alongside vector search, not instead of it? If no: exact-match queries (identifiers, specific values, proper nouns) are silently failing. The agent receives an approximate answer and doesn't know to doubt it.

4. Are you applying a cross-encoder reranker over the top candidates before injecting into context? If no: your top-three results are determined by approximate similarity scoring. The most relevant chunk may consistently land at position four.

5. Have you ever inspected the actual chunks injected into context for a task that produced a wrong answer? If no: you're debugging agent behavior without looking at the inputs. The cause of a wrong answer is almost always visible in the retrieved context that preceded it.

6. Can your retrieval system distinguish a chunk from an architectural decision record versus a code comment versus a deprecated document? If no: all sources are equal in your index. Authority and status are invisible, and the agent has no basis for preferring a current architectural decision over a deprecated code comment that says the same thing differently.

7. Do you have instrumentation that logs what was retrieved for each agent task, so you can audit retrieval quality when tasks go wrong? If no: retrieval is a black box. Debugging requires reconstruction from symptoms rather than direct inspection of inputs.

Four or more “yes” answers indicates a retrieval system doing the work. Fewer than four means there’s a precision problem that no amount of prompt engineering on top of it will compensate for.

Retrieval keeps the context window from bloating with information that was speculatively pre-loaded. But retrieval has a cost: every chunk injected is a chunk that persists in context for the remainder of the session. Tool outputs accumulate turn by turn. Retrieved documents stack. Even a well-engineered retrieval system feeds a window that’s actively growing. The third discipline handles what happens after the window starts filling up, and before that growth starts degrading the session.

Compress: Shrinking What You've Already Loaded

> "Alex would run on our trace and span data. The spans would grow. There would be too much data. We'd hit a context limit. Alex would fail and the span has that data. So it would try again. We'd add more data to it. It would run and then it would fail."
>
> source: Sally-Ann Delucia, Head of Product at Arize (source: Sally-Ann Delucia (Arise), "Hierarchical Memory: Context Management in Agents")

The context window doesn't fill from the top. It fills continuously: every tool call, every retrieved chunk, every model response adds tokens to the tail. Even a perfectly engineered Generate and Retrieve layer produces a context that accumulates with each turn. Delucia's loop above is what accumulation looks like when nobody designed for the growth.

Compress is the third discipline of context engineering: reducing what's already in the window without losing the signal that still matters. It's not about taking less in. It's about managing what's already there before it becomes noise.

---

the economics of the KV-cache

Compression isn't about saving words. It's about managing a specific technical resource with concrete economic properties: the key-value cache.

When a model processes a prompt, it converts each token into key-value pairs that encode that token's representation in the attention mechanism. If the prefix of a prompt is identical to the prefix of a previous call, the model reuses the cached KV pairs instead of recomputing them. This cache hit is the mechanism behind the dramatic latency and cost differences between first-call and subsequent-call processing in production systems.

The economic consequence is stark. Manus's work on KV-cache efficiency shows that a single-token change at the start of a prompt can invalidate the entire cache, producing a 10x spike in latency and cost per call (source: AgentSight (arXiv:2508.02736)). That's not a marginal effect. In an agentic

loop that makes 50 to 100 tool calls per session, a cache that's continuously being evicted because the system prompt is mutable means every call pays full recompute cost. An agentic loop where the prefix is stable pays full cost once and amortizes across every subsequent call.

This means the structure of your context (what goes first, what changes, what stays constant) has direct dollar consequences. Context engineering isn't just about correctness. It's about cost.

The Manus pattern formalizes this into two principles: **stable prompt prefixes** and **append-only contexts** (source: AgentSight (arXiv:2508.02736); Kgent). The system prompt, your `CLAUDE.md` (the project-level instruction file that defines the agent's persistent rules and constraints), and any persistent architectural context go first and stay fixed across the session. New tool outputs, retrieved chunks, and conversation turns are appended at the tail: they grow the context, but they don't disturb the prefix. This maximizes cache reuse on everything that doesn't change while accommodating the growth of everything that does.

Work on eBPF-based observability for agents identifies a second consequence of unstable prefixes: "stale-context hallucinations," where an agent performs operations based on a cached world model that no longer reflects actual system state because the cache didn't invalidate cleanly when it should have (source: AgentSight (arXiv:2508.02736)). The KV-cache is simultaneously a performance asset (reuse it aggressively) and a coherence liability (a stale cache produces confident wrong answers). Stable-prefix architecture resolves this tension: you design what's stable to actually be stable, and you make mutation only happen in the tail where it's expected and explicit.

---

what stable prefixes require in practice

A stable prefix is only stable if you never touch it. That constraint is harder to maintain than it sounds, because the natural instinct when building agent systems is to inject dynamic state into the system prompt.

The common failures:

Timestamp injection. Adding the current date/time to the system prompt on every call. This mutates the prefix on every call and continuously evicts the cache, for information that should be in the conversation turn, not the system prompt.

Progress tracking inline. Embedding task status, current step number, or file paths being worked on into the system prompt. This information is mutable, it changes as the task progresses, and it doesn't belong in a stable prefix.

Mid-session rule updates. Modifying a constraint in the `CLAUDE.md` equivalent during a session because you discovered the original was wrong. This is architecturally correct behavior that has a cache cost. The right approach is to append the correction at the tail, explicitly stating it supersedes the earlier instruction, rather than editing the prefix.

Three design rules that maintain prefix stability:

Mutable state belongs in the tail, not the head. Timestamps, task IDs, current file state, progress markers: these are session-local state, and session-local state goes at the tail where it can grow and be compressed without disturbing the prefix.

`CLAUDE.md` changes happen between sessions, not within them. If a rule needs updating, update it before the session starts. If you discover a constraint needs to change mid-session, append the correction as a tail entry. The question "should the prefix be mutable mid-session" almost always has the answer no.

Make cache breakpoints explicit. When a significant context event occurs, such as a major file change or a new architectural constraint discovered through a tool call, place it at a named boundary rather than inserting it into the middle of the history. The model checkpoints its world model against the new state, the prior cache is preserved up to that boundary, and the session history has a legible structure.

---

summarization checkpoints: when to compress and how

Stable prefixes preserve the cache. They don't solve the tail. Every tool call appends its output. Every retrieved chunk appends its content. Every conversation turn appends the exchange. After enough turns, the tail is the bottleneck: long, dense, and growing.

Summarization checkpoints address this. At a defined threshold, when context exceeds a percentage of window capacity, a summarization step replaces a portion of the tail with a compressed representation.

Jared Zoneraich describes the Claude Code implementation: when context reaches roughly 92% of capacity, it compresses the middle portion, dropping verbatim history and replacing it with a summary, while preserving the head (system prompt and early architectural context) and the most recent turns (source: Jared Zoneraich (PromptLayer), "How Claude Code Works"). Head-tail preservation reflects a specific insight about where value is distributed in a long session. The head contains the invariants that govern the whole session. The recent tail contains the active working context for the current task. The middle is mostly transition: tool calls that completed, file reads that were processed, decisions now encoded in the current state. The middle can be compressed without losing the signal the agent needs to continue.

Sally-Ann Delucia's team arrived at a structurally similar pattern independently, with an important addition: the truncated middle is stored in a retrievable memory store, not discarded (source: Sally-Ann Delucia (Arise), "Hierarchical Memory: Context Management in Agents"). The agent can retrieve any stored portion if a subsequent task needs it. This transforms truncation from a lossy operation into a tiered storage operation: the information isn't gone, it's been moved from Tier 1 (active context) to Tier 2 (session memory). The retrieval mechanism remains available; the context budget cost is eliminated.

LLM-based summarization (asking the model to compress the middle into a prose summary) is tempting and consistently disappointing. Delucia's team tried it before smart truncation and found it "too inconsistent": the model decided what was important to keep and consistently missed decision-critical state while preserving narrative flow (source: Sally-Ann Delucia (Arise), "Hierarchical Memory: Context Management in Agents"). The model is optimizing for a compressed narrative; what survives is the story of what happened, not the constraints that should govern what happens next. For most compression cases, structured truncation with retrieval access outperforms LLM summarization because it preserves the source rather than substituting the model's interpretation of it.

Use LLM summarization selectively: for human-readable session notes where a coherent narrative is the goal, for genuinely narrative content where meaning survives compression well, and for cases where exact source preservation is impossible (real-time data streams with no persistent store). Use structured truncation with retrieval access for everything else.

---

append-only logs as first-class architecture

The append-only constraint is more than a caching optimization. It's an architectural commitment that makes context management predictable and auditable.

When context only grows at the tail, you always know where the boundary between stable and mutable is. Compression events have a defined target (the middle of the tail). Cache invalidation has a defined trigger (changes before a specified position). Debugging has a clean timeline: what was known at turn one, what was added at turn ten, what the agent decided at turn 15 and why. You can reconstruct the agent's world model at any point in the session by reading the context up to that turn.

Compare this to a context history with edits, insertions, and updates at arbitrary positions. The agent's world model at turn 15 depends not just on what was in context but on which version of previously-stated facts was current, an audit that requires replaying the mutation history, which most systems don't preserve.

Versioned Generic Context Optimization (VGCO) extends the append-only pattern to tool documentation: it uses an LLM-as-editor to automatically refine how tools are documented based on real failure cases, reducing the ambiguity between how tools are described for humans and how models interpret them. The VGCO mechanism is hierarchical and state-aware: it optimizes the documentation that lives in the stable prefix, which means each improvement compounds across every subsequent session that reuses the cache. An improved tool description in the prefix is a permanent improvement; the cache reuse ensures you pay for it once and benefit from it for the lifetime of that documentation.

Jared Zoneraich recommends instructing agents to explicitly save significant intermediate state to files rather than carrying it in context (source: Jared Zoneraich (PromptLayer), "How Claude Code Works"). This is the append-only pattern applied to the file system: the agent externalizes its working state rather than accumulating it in the window. A task that requires maintaining a running list of discovered dependencies, or a map of edited files, or a log of approaches tried: these are all candidates for file-backed state rather than context-window state. Files are persistent, cheap, and don't consume tokens when not referenced.

---

compress versus evict: choosing the right operation

Compress and evict are not synonymous. The distinction matters because the wrong choice for a given situation either wastes tokens or loses information the agent still needs.

Compress reduces the size of information in the active context while preserving access to it. The information still exists, either in a summary or in a retrievable memory store. The agent's world model retains access to it; the cost is lower resolution or the friction of an explicit retrieval call.

Evict removes information from the active context entirely. If the information isn't persisted to a secondary store, it's gone for this session. The agent can't recover it without starting the source over.

The decision:

Compress when the information might still be relevant to tasks the agent hasn't started. A prior tool call's output that informed a decision might be needed again if the agent doubles back. Compress to Tier 2 (session memory), not evict.

Evict when the information is now reflected in persistent state: the file was written, the database row was updated, the decision was made and recorded. The context no longer needs to carry it because the real-world state carries it. The agent can re-read the file if it needs the information again. Eviction in this case is not a loss; it's a cleanup.

Evict when the information is superseded. A prior retrieved chunk about a deprecated approach, a version of a rule that was updated mid-session, an earlier draft of a file that was then rewritten: these are taking up tokens for a world model that's no longer current.

Peter Werry's explicit warning against caching answers applies here: if you evict raw context and store the inferred answer in its place, you're building on a derived artifact rather than source truth (source: Peter Werry (Unblocked), "Mergeable by default: Building the context engine to save time and tokens"). The answer was correct given the context at the time. If the context changes (a file was updated, a constraint was added) the cached answer is wrong and the system has no mechanism to detect this. Evict the raw context; preserve the ability to retrieve and reprocess fresh context if the question arises again.

---

the hierarchical memory model

Compression makes sense within a four-tier architecture that defines where information lives at each stage of an agent session:

```
Tier 1: Active context window   — what the model sees right now
                                  (prefix + active tail)

Tier 2: Session memory store    — truncated middle, retrievable on demand
                                  within this session

Tier 3: Long-term storage       — files written to disk, database state,
                                  anything that persists across sessions

Tier 4: Knowledge base          — indexed, chunked, retrievable via
Retrieve
                                  discipline
```

A compress operation moves content from Tier 1 to Tier 2. An evict operation moves it from Tier 1 to Tier 3 or Tier 4, or drops it entirely if it no longer has value.

The practical implication: don't design compression in isolation. Design the full tier system. Every piece of information that enters Tier 1 should have a defined lifecycle: how long it stays in Tier 1, what happens when it needs to compress (Tier 2 or summarize?), whether it belongs in Tier 3 after the session (write to file?), whether it belongs in Tier 4 long-term (index as a knowledge artifact?). An agent with no tier architecture defaults to accumulating everything in Tier 1 until the window fills. That's the vicious loop Delucia described: growth with no controlled exit path.

Sally-Ann Delucia's team discovered that for longer and longer conversations, users pushing sessions to 20+ turns, even smart truncation needs long-term memory to avoid starting fresh on each new session (source: Sally-Ann Delucia (Arise), "Hierarchical Memory: Context Management in Agents"). Tier 2 handles within-session continuity. Tier 3 and 4 handle cross-session continuity. Compression is the mechanism that moves information between tiers in both directions.

---

long-session evals: the only reliable signal

The most common error in compression strategy design is evaluating performance at turn one or turn five. Compression problems are late-session problems. They manifest at turn 20, turn 30, turn 50, after the context has been through multiple compression events and the residual structure of what survived starts governing what the agent can still reason about.

Delucia's team recognized this and built long-session evals specifically to catch it: load 10 turns, test the 11th turn for context continuity (source: Sally-Ann Delucia (Arise), "Hierarchical Memory: Context Management in Agents"). The test asks: does the agent still have accurate access to information established early in the session? Does it remember a constraint stated at turn two when answering at turn 15? Can it correctly reference a prior decision when the question comes up again?

These evals expose compression failures that short sessions hide:

- A decision was compressed into summary and the summary lost the specific constraint, not the general direction.
- A retrieved chunk was evicted and the agent re-retrieved a different, conflicting chunk on the same topic.
- The prefix was subtly altered by a mid-session injection, evicting the cache and causing the agent to re-read its own instructions with slightly lower precision.

Without long-session evals, you're flying blind on exactly the failure mode that compression is supposed to prevent.

---

worked example: compressing a tool output

A GitHub pull request review tool returns the following raw response when the agent fetches PR #1842:

```
{
  "number": 1842,
  "title": "Replace session-cookie auth with JWT bearer tokens",
  "state": "merged",
  "merged_at": "2026-05-19T14:23:00Z",
  "merged_by": { "login": "nadia-k", "id": 88214 },
  "base": { "ref": "main", "sha": "a3f9c12..." },
  "head": { "ref": "feat/jwt-auth", "sha": "d7b4e88..." },
  "additions": 412,
  "deletions": 187,
  "changed_files": 23,
  "body": "Replaces session cookie auth with RS256 JWT bearer tokens...
        \n[347 words of PR description]\n\nBreaking changes:\n-
        req.session.userId → req.auth.sub\n- Middleware now rejects
        expired tokens with 401\n\nMigration: docs/auth-
        migration.md\n\nTested on staging. All e2e tests passing.",
  "labels": [{ "name": "breaking-change" }, { "name": "auth" }],
  "requested_reviewers": [...],
  "review_comments": 14,
  "comments": 6,
  "commits": 8,
  "diff_url": "https://github.com/.../pull/1842.diff",
  "patch_url": "...",
  "html_url": "..."
}
```

That response is roughly 200 tokens. The agent needs it to answer: “what changed in auth last week?” Most of the payload is irrelevant to that question.

Compressed to 30 tokens:

```
PR #1842 (merged 2026-05-19): session-cookie auth replaced with
JWT RS256. Breaking change: req.session.userId → req.auth.sub.
Expired tokens now return 401. Migration guide: docs/auth-migration.md.
```

What was lost:

The contributor name, commit SHA, additions/deletions count, reviewer list, comment counts, diff and patch URLs. None of that governs how the agent should write new code. The PR description body (347 words) collapsed to two sentences because the breaking change section was the only load-bearing content.

The loss is fine because:

The compressed summary contains every constraint the agent needs at the moment of use. If the agent later needs the exact migration guide, it can fetch `docs/auth-migration.md` directly. If it needs the diff, the `diff_url` was not in

the summary but the PR number was: the agent can reconstruct the fetch. The source is preserved in Tier 3 (the git history). The compressed summary is a Tier 1 representation that serves the current task without consuming 170 unnecessary tokens on every subsequent turn.

This is the structured extraction pattern: identify which fields the downstream task actually needs, inject only those, discard the rest. Write the extraction transform once per tool type. Every subsequent call of that tool type produces a compact, task-relevant injection.

---

diagnostic: is your compression working?

Run this check when sessions consistently underperform as they grow longer, or as a proactive audit before you observe the problem.

1. Does your system prompt or `CLAUDE.md` equivalent change between tool calls in the same session? If yes: you're continuously evicting the KV-cache. The 10x cost penalty applies to every call after the first. Restructure so the prefix is genuinely stable.

2. Do you have a defined threshold at which compression (truncation or summarization) triggers? If no: compression is either not happening or happening at the model's discretion when forced by context limits. Reactive compression happens at the worst possible moment, when the agent is already degraded.

3. Is the truncated or summarized content stored in a retrievable session memory, or is it discarded? If discarded: truncation is eviction, and information needed later in the session is permanently lost. Store the middle; give the agent a tool to retrieve it.

4. Are tool outputs injected verbatim into context, regardless of their size? If yes: every large tool response is a compression problem. A 50KB JSON response that contained three relevant fields is 45+ KB of context debt. Summarize or extract before injection.

5. Are agents writing significant intermediate state to files rather than accumulating it in context? If no: working state that could live in Tier 3 is consuming Tier 1 budget. Files are persistent and token-free when not referenced. Treat file writes as explicit context offloading.

6. Do you run evals that test context continuity at turn 15 or later, not just at session start? If no: your compression strategy has never been tested against the failure mode it’s supposed to prevent. Build a long-session eval set before you observe the failure in production.

Four or more “yes” answers means compression is a managed process. Fewer than four means context growth is passive and uncontrolled, and session quality will degrade as a function of session length.

Generate, Retrieve, and Compress together manage the information that flows into, through, and accumulates within a single agent’s context window. But complex work, work that spans multiple concerns, multiple files, multiple agents operating in parallel, can’t be solved by managing one window better. It requires a different kind of management: deciding which work goes to which context, and ensuring each context stays focused on its assigned piece of the problem.

Route: The Rules That Actually Apply Right Now

Your `CLAUDE.md` (the project-level instruction file that governs how your AI agent behaves) has grown to 44,000 characters. You added a rule every time the model made a mistake. The rules are all correct. And the model still ignores them.

The problem isn't the rules. It's that all of them arrive at once.

Predicate-gated rule injection is the discipline of deciding which rules enter the context for a given prompt, not which rules exist in your system. The rule set grows without limit; the injected set stays bounded.

---

the injection problem at scale

A flat instruction file works at 10 rules. At 30, it becomes background noise. At 44,000 characters, it consumes your context budget before the first user message arrives.

The failure is predictable. You started with a tight `CLAUDE.md`. You caught the model making a mistake and added a rule. The rule worked. The model made a different mistake; you added another rule. This process is correct behavior: you're encoding operational knowledge. The problem is architectural: every rule you add is re-injected on every prompt, regardless of whether that rule could possibly apply.

A rule about Python type annotations fires when you're editing a Dockerfile. A rule about commit message format fires when you're mid-session in a feature branch with no git operations pending. A rule about never using `sudo` in production fires when you're in a read-only research conversation. All of them arrive together, all of them compete for attention, and the model treats the block as boilerplate because, statistically, most of it is.

The solution isn't fewer rules. It's rules that only show up when they're relevant.

The pattern that solves this is decades old. ESLint has `paths` and `overrides` per rule (file glob matching). Semgrep has `paths.include` / `paths.exclude` in YAML rule frontmatter. Drools evaluates LHS conditions and only matched rules enter the agenda. OPA / Rego rules return undefined when input doesn't match, contributing nothing. MegaLinter auto-detects file types and activates only matching linter packs.

The shared insight across all of these: at scale, the runtime decides which rules apply. Rules ship with self-describing activation conditions. The engine pre-filters.

The same architecture works for LLM rule systems. You build a rule vault that can grow indefinitely. Each rule carries a predicate. The runtime evaluates predicates against the current execution context and injects only matching rules. Total injected size stays bounded by situational match, not vault size.

---

predicate styles and the always-on core

There are two mechanisms for deciding whether a rule fires. They solve different cases and work better together than either works alone.

Hard filters are deterministic, mechanical matches on observable context fields. A rule fires if `tool_name == "Bash"`. Another fires if `command_match` contains "git commit". Another fires if `file_extension == ".py"` or `cwd_contains == "aaOS"`. These are cheap, fast, and predictable.

Hard filters miss the cases where the relevant situation arrives without the expected surface signals. "We should ship this today" arrives with no git command, no commit mentioned, but every rule about release hygiene applies. Hard filters won't catch it.

Semantic matching covers the gap. Each rule carries a trigger description, a natural-language phrase describing the space of situations where the rule applies. The runtime computes an embedding cosine similarity between the current prompt and each rule's trigger description. Rules above a similarity threshold are included. ChromaDB with `all-MiniLM-L6-v2` embeddings is a practical stack for this; if you already use wiki-search with semantic indexing, you have the infrastructure.

The two styles are not competing. A rule can have hard filters AND a semantic trigger description. Hard filters are cheap first-pass gates. Semantic matching catches the residual. Combined, they provide the coverage of always-inject with a fraction of the token cost.

Not every rule should be predicate-gated. A small set needs to fire unconditionally: response style rules, output discipline rules, and safety rules that apply regardless of task. These are tagged `tier: always` or live in an `_always/` subdirectory. The production system described in this guide runs four always-on rules. That's the right order of magnitude.

The test for promotion to always-on is strict: if the rule is violated, is the violation harmful regardless of task context? If yes, it's always-on. If violation is only harmful in specific contexts, it stays gated. Most rules fail this test. A rule about never using certain variable names might feel universal, but it only applies when writing code. That's a predicate, not an always-on condition.

Keeping the always-on set at four to six rules is not an arbitrary constraint. It's the size at which the model can treat each rule as a genuine instruction rather than as part of a boilerplate block. When the always-on set grows to 20, it exhibits the same signal-dilution problem as a large flat `CLAUDE.md`, just a smaller version of it.

---

organizing the vault: folders and relevance over frequency

Predicates work at two levels. Fine-grained predicates operate at the individual rule level: specific tool names, command patterns, semantic triggers. But you can also apply a coarser structural filter by organizing rules into folders that correspond to task domains.

The production rule-router uses folder structure as a first-pass category filter. Rules in `git/` are only candidates when git operations are in scope. Rules in `vault/` are only candidates when wiki-capture or schema operations are in scope. Rules in `output/` govern response style and are candidates on every prompt. Rules in `tooling/` cover tool selection and are candidates when the model is deciding which tool to call.

Folder membership is the cheapest predicate you can author. It requires no frontmatter, no embedding, and no semantic matching. The router knows the task domain and can exclude entire folders before evaluating individual rules. A debugging session that never touches git doesn't even load the `git/` folder; the rules in that folder are invisible for that prompt.

When your rule vault grows, organize by category before you invest in fine-grained predicates. Category-level organization eliminates the largest fraction of irrelevant rules with the least authoring overhead. Fine-grained predicates then operate on a much smaller candidate set.

An alternative to predicate-gating is frequency-based tier scoring: track which rules fire most often and promote the high-frequency ones to a priority tier. Low-frequency rules get deprioritized. The logic seems sound: important rules fire often; unimportant rules fire rarely. This approach fails on a fundamental category error. Frequency measures how often a situation arises, not how important the rule is when the situation does arise.

A rule about production database migrations might fire three times a year. When it fires, it's the most important rule in the system. A model that treated it as low-priority could produce a catastrophic schema change. Frequency-based deprioritization would demote this rule exactly when you can least afford it.

The right axis is situational relevance, not historical frequency. A rule is relevant when its situation is present. Predicates encode that directly: the rule defines its own situation, and the router evaluates whether that situation is current. A rule can be dormant for 50 sessions and then be the only rule that matters on session 51. Frequency is a measurement; situational relevance is a specification.

This is the same insight that makes linter configurations work correctly. ESLint's `overrides` don't track how often a rule fires on your codebase. They track which file patterns the rule applies to. The rule about TypeScript strict null checks applies to `.ts` files, not to `.js` files, not because TypeScript files are more common, but because the rule is situationally applicable to TypeScript and inapplicable to JavaScript.

---

the rule-router daemon pattern

The architecture above can be implemented at varying levels of sophistication. The most capable pattern is a **rule-router daemon** (a small local service that evaluates predicates on each prompt and injects only the matching rules): it runs at session start and on each prompt, evaluating predicates and building the injected rule set dynamically.

The production rule-router running in this guide's reference system (described below) works as follows:

- Every prompt submission fires a `hardrule-reinject` hook that calls the daemon at `http://127.0.0.1:8765/route`.
- The daemon always injects the `_always/` rules (the always-on core).
- It then runs keyword-regex matching and semantic-cosine routing across the remaining rule vault, injecting up to three contextually matched rules per prompt.
- It falls open to legacy `CLAUDE.md` extraction if the daemon is down (logged to `/var/log/rule-router-hook.log`).

The result: every prompt receives the always-on core plus a small situationally relevant set. The total injected block stays bounded regardless of how large the vault grows. A vault of 50 rules produces the same injected size as a vault of 10 rules, since the match set is bounded, not the vault.

The `/healthz` endpoint reports loaded rule counts so you can monitor that the daemon is healthy without inspecting the vault directly.

You don't need a full daemon to start. The simplest implementation is a hook that reads the current tool call and prompt, evaluates a YAML frontmatter predicate on each rule file in the vault, and prepends the matching set to the system prompt. That's a few dozen lines of Python and provides the core capability. The daemon pattern adds semantic matching, ChromaDB integration, health monitoring, and fallback behavior, useful once the vault grows past 20 rules and the predicate logic is worth centralizing.

The bounded-injection design is what makes the rule-router's behavior qualitatively different from a large flat `CLAUDE.md`. When a developer writes 44,000 characters of instructions, the model receives all of it on every prompt. At that scale, the model treats the block as boilerplate; statistically, most of any given rule block is irrelevant to the current prompt. The model has no

basis for distinguishing the three rules that apply right now from the 40 that don't. The rule-router makes that distinction before the model ever sees the injected content. The signal-to-noise ratio stays high independent of vault size.

The meta-adaptive context engineering framework from Jointly extends this principle into a learned meta-controller that profiles each task and allocates the right combination of strategies. The insight is the same: instead of applying the same procedure to every problem, profile first, then match. (source: Alberto Romero (Jointly), "The Unbearable Lightness of Agent Optimization") A rule-router daemon is a simpler instance of the same architecture: predicate evaluation is the profiling step; rule injection is the strategy allocation.

This pattern generalizes beyond rules. Document pages can tag their activation context so only matching docs load. Hook matchers already implement the same idea: `tool_name: "Bash"`, `tool_name: "Edit|Write|MultiEdit"`. Predicate-gating is the general form of what hooks already do specifically.

---

authoring predicates that work

A rule is only as good as its predicate. A predicate that's too broad defeats the purpose: it fires on everything, returning you to the wall-of-rules problem. A predicate that's too narrow misses the situations the rule was written for.

Three authoring principles:

Be specific on hard filters. `tool_name: Bash, command_match: "git commit"` is better than `tool_name: Bash`. The latter fires on every bash call; the former fires only when you're about to commit. Every over-broad hard filter increases injected size and dilutes the matched set.

Be permissive on semantic triggers. The trigger description should describe the space of situations where the rule applies, not a list of literal trigger phrases. A rule about merge conflict hygiene might have a trigger description like "when resolving or preparing to resolve code conflicts between branches." That covers "git merge", "merge PR", and "we have a conflict in this file," none of which are the literal phrase, all of which are within the situation.

Author the predicate at capture time. When you write a new rule, write the predicate immediately. Rules authored without predicates get added to the always-on block by default, which is precisely the bloat you're trying to avoid. The friction of writing the predicate is what keeps the always-on core small.

A concrete worked example: a rule that says "never delete files without explicit confirmation." Authored as always-on, it fires on every prompt, including read-only conversations where no file operation is possible. The model carries the instruction as dead weight. Authored with a hard filter of `tool_name: Bash|Write|Edit` and a semantic trigger of "file deletion, overwrite, or destructive operation," it fires only when a destructive tool call is imminent or suggested. Same protection. Fraction of the context cost.

One caveat on what predicate-gating does not solve: rule conflicts. If two rules contradict each other and both predicates fire on the same prompt, the engine includes both. Conflict resolution is a separate concern, usually handled by a `priority:` field in the rule frontmatter or by a most-specific-predicate-wins convention. Author your rules with that in mind, and when two rules feel like they're in tension, ask whether they should be one rule with a more nuanced body.

---

worked example: a routable rule

Here is a complete rule file as it lives in a production rule vault. The filename is `rules/git/commit-hygiene.md`.

```
---
tier: gated
trigger: "committing, staging files, writing commit messages, pushing
        code"
hard_filters:
  tool_name: ["Bash"]
  command_match: ["git commit", "git push", "git add"]
priority: 20
---

# Commit hygiene

Every commit message must have a subject line under 72 characters.

Use the imperative mood: "Add feature" not "Added feature."

Include a co-author trailer when Claude Code generates the commit:

    Co-Authored-By: Claude Opus 4.7 <noreply@anthropic.com>

Never commit directly to main. If the current branch is main,
stop and ask which branch to use.

Never force-push to a shared branch. If the remote has diverged,
use `git pull --rebase` and resolve conflicts explicitly.

After each push, confirm the remote accepted the ref:
run `git log --oneline origin/main -3` and state the result.
```

The frontmatter is what the **rule-router daemon** (the small local HTTP service that evaluates predicates on each prompt and injects only matching rules into the context) reads before the model ever sees the rule body.

`tier: gated` means this rule is not in the always-on core. It requires a matching predicate to inject.

`trigger` is the semantic description. The daemon embeds this phrase and computes cosine similarity against the current prompt. A user message that says “we should ship this today, let’s get these changes committed” has no `git commit` literal, but the semantic embedding is close enough to trigger the rule.

`hard_filters` provide the cheap first-pass gate. If the tool call is not `Bash`, or the command doesn’t contain a git operation string, the rule is excluded before the semantic step runs. This makes the rule invisible in the 95% of prompts that have nothing to do with git.

`priority: 20` is used when two rules in the same session might conflict. Higher priority wins.

A prompt that triggers this rule:

```
User: "okay I think the auth migration is done, let's commit
      this and push to the feature branch"

Tool context: Bash tool active, previous command was "git status"
```

The daemon sees `tool_name: Bash` (hard filter match), and the phrase “commit this and push” has high cosine similarity to the trigger description. The rule is injected. The model sees the commit hygiene body before generating the commit command.

A prompt that does not trigger this rule:

```
User: "can you review the auth migration changes before we commit?"

Tool context: Read tool, no Bash call pending
```

The hard filter on `tool_name: Bash` fails. The rule is not a candidate. The model does not see the commit hygiene body. The prompt is about reading and reviewing, not executing a commit. Injecting the rule here would be dead weight.

What the rule-router daemon actually does on the first prompt:

1. Injects the four always-on rules unconditionally (response style, output discipline, safety core).
2. Evaluates hard filters across the remaining rule vault. Most rules fail the `tool_name` check and exit.
3. Runs semantic cosine scoring on the rules that passed hard filters.
4. Selects up to three contextually matched rules above the similarity threshold.
5. Prepends the matched set to the system prompt alongside the always-on core.

The injected set for the “let’s commit this” prompt above: the four always-on rules plus `commit-hygiene.md`. Total injected: roughly 400 tokens, not 44,000. The vault can contain 100 rules. The prompt sees five.

That bounded injection is the structural property the flat `CLAUDE.md` can never achieve regardless of how well the individual rules are written.

---

routing diagnostic

Run this against your current rule system. Score one point for each “yes.”

1. Do you have more than 15 rules in your CLAUDE.md or equivalent? If yes: the flat injection model is already degrading. Your rules are competing for attention; the model treats the block as boilerplate.

2. Do any rules fire on every prompt even though they only apply in specific tool contexts (e.g., git, file editing, code generation)? If yes: you have situational rules masquerading as always-on rules. Every one of those is a predicate you haven’t authored yet.

3. Has the model ever followed an instruction correctly in one session and ignored the same instruction in another, with no apparent difference in your prompt? If yes: the difference was likely context composition. Different situational content shifted which tokens the model weighted. Predicate-gated injection makes that composition explicit rather than implicit.

4. Do you ever re-inject the same rules explicitly in your prompt because you don’t trust the model will “remember” them? If yes: you’re compensating for injection unreliability by burning context budget manually. Predicate-gating replaces the prayer with a mechanism.

5. Has your CLAUDE.md grown primarily by addition, rules added when something went wrong, with no corresponding pruning or restructuring? If yes: you have an unbounded append-only rule log. Every rule is still being injected, including rules that only apply to tools or contexts you rarely use.

Score 0–1: Your rule system is healthy. Add predicates as the vault grows. **Score 2–3:** You’re in early-stage bloat. Author predicates for the top five most-situational rules. Measure whether instruction-following improves. **Score 4–5:** You’re injecting a context-budget tax on every prompt. Restructure around the always-on core + predicate-gated vault pattern before adding any new rules.

Routing keeps the right rules in scope. But even a well-routed context window fills, not from rules, but from the accumulated output of tool calls, sub-operations, and branching research. Once the window is full of material that served its purpose and is no longer needed, you’re not facing a routing problem. You’re facing an eviction problem.

Evict: Clearing What the Session No Longer Needs

You’re 40 minutes into a debugging session. The agent has read eight files, run a dozen tool calls, and generated several hundred lines of output tracing a problem through three layers of the stack. You found the bug 20 minutes ago. Everything since then, the intermediate reasoning, the dead ends, the raw stack traces, is still in the context window, sitting there, consuming budget, competing with the instructions that matter.

Context rot (the slow decay of session quality as the token budget fills and early instructions lose attention weight) doesn’t only come from rules that drift or tool outputs that bloat. It comes from work the session already finished.

Eviction is the discipline of deciding what to remove from the context window and when, before accumulated material from prior work degrades the remaining work.

---

what prompt soup actually contains

Prompt soup is a state of context window saturation where competing instructions, fragmented history, and raw tool outputs degrade the model’s reasoning coherence.

Three categories produce the majority of the bloat:

Raw tool outputs. An MCP (Model Context Protocol, the standard interface that connects external tools and data sources to an AI agent) call returns a 50KB JSON blob. A file read returns 800 lines of source. A search returns 20 results with full snippet text. These outputs aren’t prompt content; they’re data that should be processed and summarized before they touch the context window. When they arrive verbatim, they displace the instructions and decisions that should be load-bearing.

Completed work history. Every resolved sub-task leaves its trace in the conversation. The research phase that established a fact the implementation phase is now acting on. The error messages from three failed approaches to a

problem you've since solved. The scaffolding reasoning from the first pass that the second pass corrected. All of this was necessary at the time. None of it is necessary now. But it's still there.

Contradictory steering. When something goes wrong mid-session, the instinct is to correct course by adding more instructions. "No, wait, I meant X not Y." "Undo that approach, try Z instead." Those correction tokens accumulate alongside the original (now-wrong) instructions. The model sees both. It reconciles them imperfectly, and the reconciliation degrades on every additional correction. Brendan O'Leary describes the practical consequence directly: "Old or contradictory context — like trying to steer the agent back after a wrong turn — can persist and cause regression. It's better to start a new session than to try to correct mid-stream." (source: Brendan O'Leary, "Agentic Engineering: Working With AI, Not Just Using It")

The context window is not a database with indexed retrieval. It's a fixed-size attention surface. When the window fills, the model doesn't lose access to old content the way RAM is freed; it's still technically present. What degrades is the attention weight those early tokens receive relative to later, more recent content. The architect-level decisions established at turn three compete with the mechanical back-and-forth of turn 30, and they lose that competition as the session grows. Eviction is the discipline that manages that competition intentionally rather than letting it resolve by recency bias.

Eviction addresses all three bloat categories. The mechanism differs by category.

---

compact vs. subagent-offload

There are two eviction strategies. Choosing the wrong one costs you either context budget or task continuity.

Compaction is the lighter intervention. You summarize a completed section of work (a debugging thread, a research phase, a design exploration) into a dense paragraph or structured note. The raw output goes away; the essential conclusions stay. You keep working in the same session with a reduced context footprint.

Compaction works when the completed work produced a conclusion that downstream work will reference, the session still has a clear continuation path, and the compressed summary is genuinely sufficient: you won't need to re-examine the reasoning, only apply the result.

O'Leary frames the practical signal: "AI is great at summarizing itself — ask it to summarize the session for a new agent, review, then continue with fresh context." (source: Brendan O'Leary, "Agentic Engineering: Working With AI, Not Just Using It") That's compaction applied at session boundaries. The same technique applies within a session to summarize completed phases without abandoning the current one.

The research-plan-implement loop operationalizes this directly. The research phase produces a document. The plan phase produces a step list. Then you start a new session with just the plan: the research phase output is compacted into its summary, and the implementation session begins with clean context. (source: Brendan O'Leary, "Agentic Engineering: Working With AI, Not Just Using It") Each phase evicts the prior phase's raw trace before the next phase begins. The plan is the only thing the implementation phase needs from everything that came before it.

Subagent offload is the deeper eviction. You move a work unit out of the parent session's context entirely by dispatching it to a child agent with a focused context, letting that agent complete the work, and receiving back only the artifact and a summary. The parent session never accumulates the intermediate steps.

Most engineers think of subagents as a capability feature: "my orchestrator can delegate to specialists." That's accurate, but it misses the structural reason subagent architecture works for long-running tasks. When you dispatch a subagent, you're not parallelizing work. You're evicting a work unit from the parent's context before it generates the tokens that would have accumulated there. The parent's context stays bounded at the price of a dispatch-and-return overhead. Tasks that would have saturated a single session in 40 minutes can run for hours in an orchestrator-plus-roster architecture without the orchestrator approaching context saturation.

The team-lead tier above the IC roster (the set of specialist subagents the orchestrator can dispatch work to, each handling one task type) implements exactly this pattern. When the orchestrator dispatches an `eng-lead` or `implementer`, the multi-file change, the test generation, the refactor: none of those intermediate steps live in the orchestrator's context. The orchestrator

receives a structured `SUMMARY / FILES CHANGED / VERIFICATION / DEFERRED` return shape. The full implementation trace stays in the subagent's session. The team-lead tier was introduced to solve a routing problem (the orchestrator had too many ICs to route directly) but its structural effect is also an eviction solution. Every work unit dispatched through a lead stays out of the orchestrator's context.

This is why the dispatch-and-return protocol (the structured format for sending work to a subagent and receiving a bounded result summary back) includes a `return_format` field: the parent specifies exactly how much it wants back. The return shape is an eviction budget decision: how much of the subagent's work needs to re-enter the parent's context?

The decision rule: if you need the conclusion and can discard the path, compact. If you need the work done but the doing is not the parent session's concern, subagent-offload. If the work would take more than five tool calls in the parent session, it belongs in a subagent.

---

truncation policies for tool outputs

Not every eviction decision involves sessions or subagents. The most frequent eviction decision is the smallest one: what to do with the output of a tool call before it hits the context.

The default behavior of most tool integrations is verbatim injection. The file read returns 800 lines; all 800 go into the context. The API call returns a nested JSON object; all of it lands in the model's working memory. This is the token bloat failure mode from Chapter 2, operating at the micro level.

The failure compounds with MCP servers. O'Leary documents the specific mechanism: each enabled MCP adds tool descriptions to the system prompt, even if the MCP is not being used. A Postgres MCP enabled during front-end work wastes tokens and can confuse the agent into touching the database when no database operation was requested. The tool description occupies context budget whether or not the tool fires. (source: Brendan O'Leary, "Agentic Engineering: Working With AI, Not Just Using It")

Four truncation policies, in ascending order of invasiveness:

Length cap with summary. Tool outputs above a threshold are truncated, and the truncation point is accompanied by a summary of the omitted content. The model sees the first N lines plus "…truncated; remaining 600 lines

contain Y.” A pure character limit without a summary creates a silent gap: the model doesn’t know what it didn’t see. The summary is what makes the truncation non-lossy from the model’s perspective.

Structured extraction. Instead of injecting a raw tool output, extract the fields the downstream work actually needs. An API response that contains `name`, `id`, `status`, `created_at`, and 40 other fields: if the current task only needs `status`, inject `status` only. Every unneeded field is a token you don’t pay. Structured extraction is most effective for tool outputs with stable schemas. Write the extraction transform once, and every call of that tool type produces a compact, task-relevant result.

Append-only summary log. Tool outputs are never injected verbatim. Each call appends a structured entry to a running log: timestamp, tool, result summary, conclusion drawn. The log is what lives in the context. When the log itself grows too large, compact the older entries. This is the most disciplined form: it requires authoring the summary at injection time rather than letting the model process raw output. The upfront cost is real, but the downstream benefit is a context window that degrades slowly and predictably rather than in sudden bloat spikes when a large tool output arrives.

Session isolation. Some tool calls produce outputs that are load-bearing for one sub-task and irrelevant afterward. Search results used to orient a research phase, for example. Session isolation treats the tool call and its output as belonging to a dedicated sub-session: dispatch a subagent, let it process the raw output, receive back the synthesized conclusion. The parent never sees the raw output. The subagent’s context saturates on the large tool output; the parent’s context receives only the extracted signal.

The right policy depends on how stable the output is and how often the downstream work needs to re-reference it. A search result used once to orient a phase: session isolation. A configuration file referenced across many subsequent steps: structured extraction. An API response that accumulates new data over the session: append-only log. A large file read needed only for a narrow inspection: length cap with summary.

Disabling unused MCPs is the zeroth policy, before any of the above. If a tool isn’t needed for the current session, its description shouldn’t be in the context at all. The context budget is a shared resource. Every tool description that occupies that budget is a cost paid on every prompt regardless of whether the tool fires.

---

the over-eviction failure mode

Eviction can fail in both directions. Under-eviction produces the prompt soup described throughout this book. Over-eviction produces a different failure: the agent loses track of why it started.

The classic over-eviction mistake: compacting too aggressively during a complex debugging session. You compress the intermediate reasoning into a paragraph. The paragraph captures the conclusion ("bug is in the serializer") but drops the constraint that ruled out three other approaches. Later in the session, the agent revisits one of those ruled-out approaches. The constraint that eliminated it was in the evicted material. The session re-does work it already did.

Over-eviction also shows up at the subagent boundary. The orchestrator dispatches a subagent, the subagent does the work, the return summary is terse. The orchestrator continues. Three dispatches later, it needs context from that earlier subagent's work that wasn't in the summary. The full context is gone; the subagent session is closed. The information is either reconstructible at cost or permanently lost.

Two defenses:

Preserve the constraint log, not the reasoning. When compacting, the material worth keeping is not the narrative of what happened but the set of constraints that shaped the path. "Tried X, failed because Y" is a constraint. "The reasoning in approach X was flawed because…" is narrative. Compress the narrative; preserve the constraints. Constraints are what prevent re-doing work.

Author the return summary at dispatch time, not retrospectively. When you write the dispatch spec for a subagent, specify the return format precisely: what information the parent session will need to continue. The subagent's summary is written to that spec. If you don't know what you'll need until after the work is done, the return summary will be optimized for completeness rather than for the parent's actual continuation needs, and it will be either too long (defeating the eviction purpose) or too short (causing information loss).

Context management is a skill that improves with practice. O'Leary frames it plainly: "The art and science of working with AI comes from experience — learning what to trust and what not to trust." (source: Brendan O'Leary,

“Agentic Engineering: Working With AI, Not Just Using It”) Eviction judgment (knowing when a section of context has served its purpose and can be compacted or offloaded) is exactly this kind of experiential calibration.

The over-eviction failure is less common than under-eviction but harder to diagnose. Under-eviction shows up as degraded output or a context limit hit. Over-eviction shows up as duplicated work or a session that can’t explain why it made the decisions it made three dispatches ago. If you’re debugging an agent that’s repeating itself, check for context rot. If you’re debugging an agent that’s contradicting decisions it made earlier, check what was evicted and when.

---

worked example: a subagent dispatch and return

The parent orchestrator needs to implement a new API endpoint. It has already established the architectural context (auth pattern, error handling standard, route structure) in the first few turns. It knows if it runs the implementation work itself, the intermediate tool calls (file reads, edits, test runs, lint cycles) will consume 2,000+ tokens that the orchestrator’s context doesn’t need to carry. The work unit belongs in a subagent.

The parent’s dispatch brief (what the orchestrator sends):

```
## Dispatch

goal: Implement POST /api/orders/bulk-update endpoint.

constraints:
  - Auth: JWT bearer, validate with req.auth.sub (see PR #1842).
  - Error handling: route to Sentry, not console.error.
  - Data access: React Query hook, never call DB from component.
  - File: src/api/routes/orders.ts
  - Tests: add one happy-path integration test.

success_criteria:
  - Endpoint accepts array of { orderId, status } objects.
  - Returns { updated: number, failed: [] } shape.
  - Auth rejects requests with expired tokens (401).
  - Test passes.

return_format: structured_report
escalate_if: schema change required (out of scope).
```

That brief is roughly 110 tokens. It carries everything the subagent needs and nothing the orchestrator needs to retain once the work is done.

The subagent's full return (what comes back to the parent):

```
## Report

outcome: DONE

summary:
  - Implemented POST /api/orders/bulk-update in src/api/routes/orders.ts.
  - Auth guard applied via jwtMiddleware; expired tokens return 401.
  - Returns { updated: number, failed: Array<{orderId, error}> }.
  - Integration test added: orders.integration.test.ts, passes.

artifacts:
  - src/api/routes/orders.ts (modified, +47 lines)
  - src/api/routes/orders.integration.test.ts (new, +38 lines)

open_risks:
  - The bulk update has no rate limit. High-volume callers could
    trigger DB contention. Consider adding express-rate-limit.

escalations: none
```

That return is roughly 100 tokens.

What the parent retains:

The dispatch brief (110 tokens) plus the return report (100 tokens). Total: 210 tokens added to the orchestrator's context for a work unit that consumed roughly 2,400 tokens in the subagent's session (eight tool calls: two file reads, three edits, one test run, one lint pass, one verification).

The 2,200-token difference is the eviction. The orchestrator never saw the raw file reads, the intermediate edit attempts, the lint errors and corrections, or the test output. It received the conclusion: what was built, where it lives, and one open risk to track.

What would have happened without subagent offload:

All eight tool calls run in the orchestrator's session. The raw file reads (600+ lines of `orders.ts` and surrounding context) land verbatim. The edit iterations accumulate. By the time the endpoint is working, the orchestrator's context

has absorbed the full implementation trace. Subsequent dispatches (the next endpoint, the auth review, the migration) each start with a context already heavy with the previous work unit's debris.

The subagent boundary is the eviction operation. The dispatch brief specifies what information the parent needs back. The subagent's working context stays in the subagent's session. The parent's context receives only the structured report. The return shape is not a courtesy format. It's an eviction budget decision: exactly how much of the subagent's work re-enters the parent.

---

eviction diagnostic

Run this against your most recent long session. Score one point for each "yes."

1. Did you reach a context limit or notice degraded reasoning before completing the planned scope of work? If yes: the session under-evicted. Accumulated material displaced the reasoning budget needed to finish.

2. Did any tool call inject raw output over 1,000 tokens without a pre-processing or summarization step? If yes: you have a tool-output bloat problem. Verbatim injection is the fastest path to context saturation.

3. Did the agent repeat an approach or revisit a decision that had already been resolved earlier in the session? If yes: you over-compacted. The constraint that ruled out the revisited approach was in the evicted material.

4. Did the session accumulate correction turns ("no, wait, I meant X, not Y") without a session reset? If yes: you have contradiction accumulation. The model is reconciling conflicting instructions. Start a new session with a clean handoff summary rather than trying to steer mid-stream.

5. For multi-step tasks, did you run all steps in a single session rather than dispatching bounded work units to subagents? If yes: your orchestrator is accumulating the work trace of every IC-level operation. The context budget should stay with the orchestrator; the work belongs in subagents.

Score 0–1: Your eviction practice is solid. Watch for the over-eviction failure as sessions grow more complex. **Score 2–3:** You’re accumulating avoidable bloat. Add structured truncation for the largest tool outputs and identify one recurring work unit that belongs in a subagent. **Score 4–5:** You’re treating the context window as infinite. The five-discipline harness doesn’t work without eviction. Start with session isolation for raw tool outputs and subagent offload for work units over five tool calls.

Generate, Retrieve, Compress, Route, Evict: the five disciplines form a closed loop, not a checklist. Generate decides what enters at session start. Retrieve surfaces the right material at the right moment. Compress reduces token cost of what’s present. Route keeps only situationally relevant rules in scope. Evict clears what’s no longer needed. With all five active, the context window works for you across the full arc of a session. Part III opens on how sessions themselves are designed before the first token arrives: what the architecture looks like from the outside.

Sandwich Architecture

Your agent finishes the task. The tests pass. The diff looks clean. You merge it. Three days later a teammate flags a function that bypasses the auth layer, writes directly to the database, and has no error handling. The agent produced working code. Code that doesn't belong.

This is the plausible-but-wrong failure mode. The model is capable enough to produce output that clears every immediate check while still violating systemic constraints it was never forced to verify against. A single prompt, no matter how detailed, can't prevent it. The prompt sets the intent; nothing confirms the output honored it.

Sandwich architecture is the pattern of structuring agent prompts into three distinct layers (a constraint slice at the top, free reasoning in the middle, and a verification slice at the bottom) so that every output is checked against the same constraints that governed its generation.

---

the working vs. belonging gap

A useful distinction: code that is "working" passes tests. Code that "belongs" aligns with the architectural invariants, naming conventions, layering rules, and tribal knowledge of the codebase it lives in. These are different bars, and most prompt engineering only addresses the first one.

The gap widens as a codebase ages. A greenfield project has few invariants. A three-year-old production system has dozens: "never bypass the service layer," "all external calls go through the client factory," "error handling follows this pattern," "these two modules must never import each other." None of these are in the code directly. They're in the heads of the engineers who built it, surfaced occasionally in PR comments, enforced inconsistently by convention. When an agent touches this codebase, it produces working code. Whether that code belongs is a different question, one the single-prompt approach doesn't answer.

Research on symbolic constraint layers for autonomous agents names this directly: the corpus converges on the conclusion that architectural integrity is a systemic property that cannot be inferred from local code smells or ensured through prompt engineering. The gap between probabilistic generation and symbolic verification is where belonging is lost. (source: Drift; COBALT-TLA)

Sandwich architecture addresses this gap structurally. It doesn't rely on the model inferring constraints from examples. It places constraints in a dedicated layer that the model reads before generating and verifies against after generating.

---

the three slices

The metaphor is deliberate. A sandwich holds its structure because the bread is on the outside. Remove either slice and the contents fall apart. The architecture works for the same reason.

The top slice: constraints. This is the system prompt layer: the invariants the output must satisfy. Not instructions for what to do, but boundaries on what's acceptable. "All database writes go through the repository layer." "Functions touching authentication must call the audit logger." "No direct imports between the billing module and the user module." Constraints are machine-checkable claims about the output, not narrative guidance about the task.

The top slice should be loaded conditionally, not monolithically. A constraint about database layering is relevant when the task touches persistence; it's noise when the task is a frontend layout change. Predicate-gated rule injection, the practice of injecting only the rules whose activation conditions match the current task, is the mechanism that keeps the top slice focused: each constraint carries an activation predicate, and the runtime admits only constraints whose predicates match the current task context. At scale, with 30 or more constraints in the vault, this keeps the top slice load-bearing rather than a wall of boilerplate the model treats as background.

The middle layer: reasoning. This is the model's working space. Once the constraints are loaded, the model reasons freely about how to accomplish the task. The middle layer is unconstrained in form: chain-of-thought, tool calls, file reads, intermediate drafts. It's the generative phase. The sandwich architecture doesn't restrict reasoning; it frames the space reasoning operates within.

The naive single-prompt approach collapses the middle and both slices into one block of text. Instructions, constraints, task description, and expected output format compete for the same attention. The model infers which parts are load-bearing. It gets it right most of the time. When it gets it wrong, treating a constraint as a soft guideline and generating output that violates it, there's no recovery step. The output ships.

The bottom slice: verification. This is the post-generation check. After the model produces output, the bottom slice runs a structured evaluation: does the output satisfy the constraints stated in the top slice? This can be a model-evaluated checklist, a symbolic lint step, or a combination. The direction matters. Verification runs after generation, against the same constraint set that governed generation. The top and bottom slices reference the same invariants; the sandwich holds because both pieces of bread are the same bread.

The orchestrator-discipline hook kit (a set of runtime hooks that enforce architectural rules automatically, blocking non-compliant tool calls before they execute) implements the bottom slice for dispatch quality: a Stop hook blocks session completion when an implementer has dispatched without a subsequent code-reviewer pass. The hook doesn't ask the model to remember the review requirement. It enforces it at the runtime level, after the generation step completes.

---

why direction matters

Top and bottom aren't interchangeable. Putting verification before generation produces a checklist the model reads and then ignores in the generative phase: "I should satisfy X" doesn't constrain output the way "I must check X after generating" does. Putting constraints after generation is post-hoc rationalization; the model is already committed to an output trajectory.

The neuro-symbolic verification approach frames this as the difference between probabilistic guidance and symbolic verification. Probabilistic guidance ("the function should follow the repository pattern") influences generation but doesn't gate output. Symbolic verification ("after generation, check: does this function go through the repository layer? If no, flag and regenerate") gates output deterministically. One is a suggestion. The other is a check. (source: Drift; COBALT-TLA)

The goal is to move the senior engineer's role from the prompt to the pipeline. The prompt can carry the task description. The pipeline, the top and bottom slices, carries the institutional knowledge that the senior engineer would otherwise have to inject manually into every PR review. Sandwich architecture automates that injection.

There's also an attention dynamic. Context windows weight recent tokens more heavily than distant ones. A constraint stated only at the top of the prompt loses attention weight as the middle layer accumulates reasoning tokens. The bottom slice refreshes the constraint's presence: the model encounters it again, in a verifiable form, after generating. Constraints stated only once, in a large prompt, are constraints that drift.

This is also why combining the verification step with the generation step in a single chain-of-thought instruction doesn't work as well as separating them structurally. "Generate the function and verify it follows the repository pattern" asks the model to self-evaluate in the same reasoning stream that produced the output. Self-evaluation in the generation stream is subject to the same trajectory bias that afflicts the rest of the output: if the generation went in a particular direction, the inline self-check tends to confirm it rather than challenge it. Placing verification in a separate bottom slice, evaluated against an explicit checklist after generation completes, produces a colder evaluation. The model isn't defending an output trajectory; it's inspecting an artifact against a list of criteria. The difference in catch rate is material on constraint violations that are subtle, violations that "feel" acceptable in the generation context but fail cleanly when checked against an explicit invariant.

---

the naive single-prompt failure

The single-prompt approach feels adequate because it often works. Describe the task, include the constraints inline, get good output. The failure mode is invisible at low complexity and becomes visible only as complexity grows.

Three dynamics compound as complexity scales:

Signal dilution. Adding more constraints to a single prompt doesn't increase constraint adherence proportionally. Past a certain density, roughly 10 to 12 constraints in a single system prompt block, the model treats the constraint section as boilerplate and applies selective attention. The constraints most directly tied to the immediate task get followed. The ones that are relevant but not visibly connected to the current output get skimmed.

No recovery path. When a single prompt produces output that violates a constraint, the violation is discovered in review, not in generation. The cost is borne by the human reviewer. Sandwich architecture moves the violation discovery to the bottom slice, before the output reaches the reviewer and before it touches a PR.

Constraint drift over sessions. A single-prompt constraint gets rewritten every time the task context changes. Over multiple sessions, the constraint language shifts: “go through the repository” becomes “use the data layer” becomes “don’t write SQL directly.” Each paraphrase is slightly different. The model applies them with slightly different strictness. Sandwich architecture keeps constraints in a canonical location, loaded identically every time their predicate matches. The constraint language doesn’t drift because it’s never rewritten.

The failure also compounds across teams. When multiple engineers work with the same agent setup, each tends to phrase constraints slightly differently in their prompts. One engineer writes “avoid raw SQL.” Another writes “use the ORM.” A third writes “go through the data access layer.” These are the same constraint. The model applies them with different strictness depending on exact wording. Canonical, centralized constraints with predicated loading remove the per-engineer variation; the constraint is the same words in the same position every time it fires.

A production system using sandwich architecture stores architectural constraints as discrete, predicated rules in the constraint vault. The top slice loader pulls only the rules whose predicates match the current task. The bottom slice runs the same rule set as a verification checklist. The middle layer reasons freely within that bounded space. The constraint list grows incrementally as new invariants are discovered. The specification bottleneck is avoided by adding constraints one at a time, as violations surface, rather than trying to define a complete formal constitution upfront.

---

worked example: a sandwich, end to end

Three files compose a real session. Read them top to bottom and the architecture becomes concrete.

Top slice: `CLAUDE.md` fragment (static, loaded at session start)

```
## Architectural constraints

- rule: no-direct-db-writes
  predicate: task touches persistence layer
  text: All database writes must go through a repository class.
        Direct ORM calls in controllers or services are a violation.

- rule: audit-on-auth
  predicate: task touches authentication or authorization
  text: Every function that reads or writes auth state must call
        `AuditLogger.record()` before returning.

- rule: no-billing-user-import
  predicate: task involves billing or user modules
  text: The billing module must not import from the user module.
        Shared types live in `packages/shared-types`.
```

These three rules load because the current task's predicate matches `touches persistence layer`. A task that touches only a React layout wouldn't load the first or third rule.

Middle layer: context excerpt (dynamic, accumulates during reasoning)

```
[turn 4, tool: Read, file: src/services/payment.service.ts]
[turn 5, tool: Read, file: src/repositories/payment.repository.ts]
[turn 6, model reasoning: The existing PaymentRepository.charge() accepts
  a userId and amount. I'll extend it with an optional metadata field...]
[turn 7, tool: Write, file: src/repositories/payment.repository.ts]
  → added: charge(userId, amount, metadata?: Record<string, unknown>)
[turn 8, tool: Write, file: src/services/payment.service.ts]
  → updated: PaymentService.processCharge calls repo.charge(...)
```

This is the generative phase. The model reasoned about what to change and wrote two files. No constraints are re-stated here; they're already in the top slice.

Bottom slice: verification step (static, fires after generation)

```
## Post-generation checklist (bottom slice)

Check each constraint whose predicate matched this task:

- [ ] no-direct-db-writes: does any new or modified function write to the
      database without going through a repository class?
      → Scan: PaymentService.processCharge calls repo.charge(). Pass.

- [ ] audit-on-auth: does any function touching auth state call
      AuditLogger.record() before returning?
      → Not applicable: this task didn't touch auth state.

- [ ] no-billing-user-import: does any modified file in the billing module
      import from the user module?
      → Scan: payment.service.ts imports from payment.repository.ts only.
        Pass.

Result: all applicable constraints satisfied. Output eligible for review
        dispatch.
```

The bottom slice references the same three constraints that were in the top slice. The session cannot close (the Stop hook blocks it) until this checklist runs and passes. The three layers are separate files: `CLAUDE.md` is static and version-controlled, the context excerpt lives only in the session window, and the checklist template is a shared artifact in the project's hook kit.

---

sandwich architecture diagnostic

Run this on your current agent setup. Score one point for each "yes."

1. Can you name three architectural invariants in your codebase that your agent doesn't explicitly know about? If yes: you have undocumented constraints that the single-prompt approach isn't enforcing. Each one is a violation waiting to be generated.

2. Does your agent prompt mix task description, constraints, and output format in a single unstructured block? If yes: your constraints are competing for attention with everything else in the prompt. Signal dilution is already active.

3. When your agent produces output that violates a constraint, who catches it: the agent or a human reviewer? If human: you don't have a bottom slice. The verification step is manual, which means it's inconsistent.

4. Do your constraints get rewritten or paraphrased across different prompts for similar tasks? If yes: constraint language is drifting. Different phrasings apply with different strictness. Canonicalize constraints in a dedicated location and load them identically.

5. Does your agent run the same constraint check on every output, or only when you remember to include it? If only sometimes: verification is aspirational, not structural. The bottom slice enforces; a note in the prompt advises.

Score 0–1: The sandwich is mostly in place. Audit the bottom slice; verification is the piece that most setups underbuild. **Score 2–3:** You have working code without belonging guarantees. Add a constraint vault for the top slice and a verification checklist for the bottom slice. Start with the three invariants you named in question one. **Score 4–5:** You're running single-prompt architecture at a complexity level that has already outgrown it. Violations exist; some have shipped. Separate the three slices before adding new tasks to the agent's scope.

The sandwich constrains what enters the context and checks what leaves it. But it still runs inside a single agent session. When tasks grow complex enough to involve multiple agents (an implementer, a reviewer, a tester) the sandwich has to scale outward as well as inward. The next problem is how to route work across a roster of specialists without the routing logic contaminating the context of the agents doing the work.

Orchestrator and IC Roster

The most common way engineers scale AI-assisted work is by asking the same session to do more. Longer tasks, more files, more steps, all handled by the main agent, which accumulates everything. After 40 minutes the context is full of intermediate reasoning, tool outputs, and completed sub-task traces. The model is technically still running. The output quality has quietly degraded.

The instinct is to use subagents (separate agent sessions that run in their own context windows and report results back to the caller) as a capability feature: “my orchestrator can do multiple things in parallel.” That’s accurate, but it misses the structural reason subagent architecture produces better outcomes on complex work. When you dispatch a subagent, you’re not primarily parallelizing. You’re keeping a chunk of work, and all the context it generates, out of the parent session’s window.

The orchestrator-plus-IC-roster pattern is a multi-agent architecture where a thin routing session (the orchestrator) dispatches bounded work units to specialized agents (individual contributors, or ICs) and receives back structured return summaries, never the intermediate steps.

---

the main session as router, not editor

The central discipline of the pattern is what the orchestrator does not do. It does not read files broadly. It does not write code. It does not run tools against the codebase. Every one of those actions generates tokens that accumulate in the main session’s context window. An orchestrator that edits files directly defeats the purpose of the architecture: the context savings belong to the IC, but the orchestrator is paying for the work anyway.

A production system enforces this with the orchestrator-discipline hook kit (a set of runtime hooks that block the main session from accumulating tool-output bloat by enforcing file-read limits and dispatch requirements). A `PreToolUse` hook on the main session denies `Read`, `Edit`, `Write`, and `Grep` calls above a per-turn threshold. When the main session tries to read its fourth file of the turn, the hook blocks the call and reminds the orchestrator to dispatch. The policy isn’t advisory; it’s enforced at the runtime level. The main session cannot accumulate tool-output bloat even if the model tries.

This matters because the orchestrator's context is the most expensive real estate in the system. It needs to hold the current task brief, the dispatch history for the session, and enough state to route the next work unit. Every file read that stays in the main session consumes budget that should go to routing decisions. Once the orchestrator's context fills with implementation traces, it loses the coherent overview that makes it useful as a router. It starts making dispatch decisions based on the details of whatever task ran last rather than the original plan.

Boris Cherny, describing Claude Code's own architecture, frames the scaling move plainly: power users run multiple Claude instances across terminal tabs or worktrees, each with its own context, coordinating through file-based handoffs. The coordination artifact, not the intermediate work, is what crosses boundaries. (source: Boris Cherny (Anthropic), "Claude Code & the evolution of agentic coding") The orchestrator-plus-roster pattern formalizes this: the structured return summary is the coordination artifact. Everything else stays in the IC's session.

---

subagent as context isolation, not parallelism

Dex Horthy's framing is the clearest available: subagents exist to control context, not to anthropomorphize roles. A subagent forks a new context window to do deep work (reading a large file, tracing a call stack through six layers, running a suite of tests) and returns a succinct summary. The parent agent stays in what he calls the "smart zone." Past roughly 40% context usage, outcomes degrade measurably. Subagents are the mechanism that keeps the orchestrator below that threshold across a long session. (source: Dex Horthy (HumanLayer), "No Vibes Allowed: Solving Hard Problems in Complex Codebases")

Brian John's implementation of subagents in Codex CLI makes the same point from a different angle. His motivation was explicit: "The main agent can give a problem to a subagent. It can go off, do its work, use its tokens, and pass just the answer back to the main agent." The phrase "use its tokens" is the key. The IC's token consumption is invisible to the orchestrator's context window. A multi-thousand-token implementation trace, a 500-line file read, a dozen tool call outputs: none of that re-enters the parent. (source: Brian John (BetterUp), "Hacking Subagents Into Codex CLI")

Context isolation compounds across a long session. An orchestrator-plus-roster session that runs for two hours and dispatches 15 work units to ICs accumulates only the dispatch briefs and return summaries in its own window. The same work done in a single session would have saturated the context window after the fourth or fifth unit. Subagent architecture isn't faster only for parallelizable work; it's the mechanism that makes long sessions possible without context rot, the slow decay of session quality as the token budget fills and early instructions lose attention weight.

---

the lead tier and the IC roster

A flat orchestrator-to-IC architecture works when the IC count is small. As the roster grows past eight or nine specialists, routing load on the orchestrator becomes cognitive overhead. Every dispatch decision lives in the main session. The orchestrator has to track what work to do, which IC to dispatch, what brief to write, and whether to serialize or run in parallel, all with a growing set of options.

The solution is a thin team-lead tier. Three leads (one for engineering work, one for quality and security review, one for research) sit between the orchestrator and the IC roster. Each lead routes only: it receives a work brief, produces a mini-plan, dispatches the appropriate ICs, collects outcomes, and returns a structured report. Leads carry no `Edit` or `Write` capability. They don't accumulate implementation traces either. Their job is routing and light coordination within their domain.

The lead tier solves a second problem beyond routing load: it owns multi-IC work units end-to-end. A feature that requires an implementer, then a test engineer, then a code reviewer is three sequential IC dispatches. Without a lead, the orchestrator has to drive that sequence, waiting for each return before dispatching the next. With the `eng-lead`, the orchestrator dispatches once with a single brief, and the lead handles the sequencing internally. The orchestrator receives one structured report covering all three ICs.

The skip-leads rule is equally important. For single-IC obvious tasks (one file to edit, one failing test to debug, one vault page to look up) the orchestrator dispatches directly to the IC. The lead tier adds latency and return-shape translation overhead. On trivial tasks that overhead exceeds the benefit. The routing rule is: multi-IC work or unclear scope goes through a lead; single-IC obvious work goes direct.

the dispatch/return protocol

The **dispatch/return protocol** is the structured format for sending a bounded work brief to a subagent and receiving a compact result summary back. Subagent architecture only produces context isolation if the return shape is disciplined. An IC that returns its full reasoning trace, every intermediate file state, and a paragraph of caveats has defeated the eviction purpose. The orchestrator accumulates exactly what it was trying to avoid.

The dispatch protocol forces specificity on what comes back. Every dispatch brief includes a `return_format` field: `structured_report`, `diff`, or `decision`. The IC authors its summary to that spec. The orchestrator specifies what it needs before the work starts, not what looks complete after the fact. Summaries written retrospectively optimize for comprehensiveness; summaries written to a pre-specified return shape optimize for what the orchestrator actually needs to continue.

The IC return shape for engineering work is fixed: `SUMMARY / FILES CHANGED / VERIFICATION / DEFERRED`. Each section is bounded. The orchestrator reads four compact sections and has everything it needs to route the next unit. The full implementation trace (the tool calls, the intermediate drafts, the test run outputs) stays in the IC session. The return shape is an eviction budget decision made at dispatch time.

The dispatch brief itself enforces quality on the other side. A production system validates briefs at the `PreToolUse` hook: any dispatch to an IC that's missing `goal`, `success_criteria`, `return_format`, and at least one of `escalate_if` or `constraints` is rejected before the subagent fires. The deny reason enumerates the missing fields. The orchestrator self-corrects and retries. Malformed briefs produce under-specified work; the hook catches them before the IC spends tokens on a task it can't complete well.

Lead return shapes follow the Report format: `outcome`, `summary`, `artifacts`, `open_risks`, `escalations`. The leads translate IC-level return shapes into this format on the way out. The orchestrator always receives the same structure regardless of how many ICs the lead dispatched internally.

model selection per role

Not all agents need the same model. The dispatch cost scales with model capability, and different roles have different capability requirements.

Review roles (code reviewer and security auditor) are the two positions where a more capable model earns its higher cost. Both roles involve catching subtle errors, applying cross-domain knowledge, and making judgment calls that affect whether code ships. A fast but less capable model reviewing a security-critical PR produces a false sense of coverage. The catch rate is what matters; the latency premium is worth it. These two roles run on the most capable available model.

Every other role in the roster, all three leads plus every executing IC, runs on a mid-tier model. Implementation, debugging, performance work, research, test generation: these are execution tasks where speed and cost matter and the quality bar is set by clear success criteria in the dispatch brief, not by open-ended judgment calls. The mid-tier model executing against a well-specified brief produces the same output as a more capable model executing against the same brief, at lower cost and latency.

Haiku-class models are excluded from the IC roster by default. The cost savings are real but the output burden offsets them. Test generation that requires a subsequent correction round, security audits that miss the obvious issue, research outputs that need full rewrites: these consume more orchestrator context and lead time than the token savings justified. The floor is the mid-tier model for all production IC work.

---

worked example: one task, three dispatches

A single feature request (“add rate limiting to the payment endpoint”) produces three sequential IC dispatches. Brief and return shapes for each follow.

Dispatch 1: to implementer

```
## Dispatch
goal: Add per-user rate limiting to POST /api/payments.
      10 requests per minute per userId; return 429 on breach.
constraints: All DB writes through the repository layer.
             Touch only src/middleware/ and src/routes/payments.ts.
success_criteria: npm test passes. 11th request within 60s returns 429.
return_format: structured_report
escalate_if: Rate limiting requires a Redis dependency not in the project.
```

Return 1: from implementer

```
SUMMARY: Added RateLimitMiddleware using in-memory sliding window.
         No Redis required; window resets on process restart (acceptable
         for v1 per the brief).

FILES CHANGED:
  src/middleware/rate-limit.middleware.ts (new)
  src/routes/payments.ts (added middleware to POST handler)

VERIFICATION: npm test: 47 passed, 0 failed.
              Manual: 11 sequential requests return 429 on the 11th.

DEFERRED: In-memory store is not shared across multiple server instances.
          Redis-backed store is a follow-up if the service scales
horizontally.
```

Dispatch 2: to code-reviewer

```
## Dispatch
goal: Review the rate-limiting implementation for correctness
      and architectural compliance.
constraints: Review only the two files in FILES CHANGED.
             Flag any no-direct-db-writes or no-billing-user-import
             violations.
success_criteria: Approval or required-changes list with line refs.
return_format: decision
escalate_if: Security concern requires architectural input.
```

Return 2: from code-reviewer

```
DECISION: Approved with one required change.

FINDINGS:
  - rate-limit.middleware.ts:34: the sliding-window map is module-scoped.
    Two test files importing the middleware share state. Add a reset()
    export for test teardown.
  - No constraint violations found.

REQUIRED BEFORE MERGE: Add reset() export. Re-run tests after.

DEFERRED: The in-memory limitation noted by the implementer is valid;
          acceptable for v1 scope.
```

Dispatch 3: to debugger (fired because reviewer flagged the test-isolation issue)

```
## Dispatch
goal: Fix test state leakage in rate-limit.middleware.ts.
      Tests share the module-scoped window map; add a reset() export
      and call it in afterEach() in affected test files.
constraints: Touch only rate-limit.middleware.ts and test files
             that import it.
success_criteria: jest --runInBand passes with no order dependency.
return_format: structured_report
escalate_if: reset() approach won't work; different isolation needed.
```

Return 3: from debugger

```
SUMMARY: Added reset() export. Called in afterEach() in two test files.

FILES CHANGED:
  src/middleware/rate-limit.middleware.ts (added reset() export)
  src/routes/__tests__/payment.route.test.ts (afterEach cleanup)
  src/__tests__/rate-limit.middleware.test.ts (afterEach cleanup)

VERIFICATION: npm test -- --runInBand: 52 passed, 0 failed.
DEFERRED: None.
```

The orchestrator’s window holds three briefs and three return summaries. File reads, reasoning traces, diff analysis, and isolation work each stayed in the IC’s session. None crossed back. The feature shipped; the orchestrator context stayed coherent for the next task.

---

orchestrator and IC roster diagnostic

Score one point for each “yes.”

1. Does your main session accumulate file reads, tool outputs, and implementation traces across a full task? If yes: your orchestrator is also an IC. Context isolation is not active; the main session is paying for work that should stay in subagents.

2. Do you dispatch subagents primarily for parallelism rather than context isolation? If yes: you're getting half the benefit. Even serial IC dispatches produce context isolation; the intermediate steps stay out of the orchestrator regardless of whether they run concurrently.

3. Can you describe what your current dispatch brief contains, specifically the `return_format` and `success_criteria` fields? If the brief is informal or varies by task: return summaries are inconsistent. The orchestrator accumulates whatever the IC thought was relevant, not what the orchestrator needed to continue.

4. For a task requiring three sequential specialists, does the orchestrator drive each step, or does a lead handle the sequencing? If the orchestrator drives each step: it's accumulating three sets of return summaries plus the routing decisions between them. A lead tier would isolate that coordination overhead.

5. Are your review roles (code reviewer, security auditor) running on the same model as your implementation roles? If yes: review quality is likely inconsistent with the coverage the review step is supposed to provide. Model selection should reflect capability requirements per role.

Score 0–1: The roster is well-structured. Check whether the return shapes are tight; summary bloat is the most common drift in mature setups. **Score 2–3:** You're getting context isolation benefits on some work units but not systematically. Formalize the dispatch/return protocol and add return_format to every brief. **Score 4–5:** The main session is running as a monolith. Dispatch latency looks like overhead, but context saturation is already the bottleneck. Add the IC boundary first, then the lead tier once the dispatch discipline is established.

The orchestrator-plus-IC-roster architecture distributes work and isolates context structurally. But the architecture depends on the agents within it following their roles. An orchestrator that edits files directly, a lead that skips

the sequencing, a reviewer that never fires: these aren’t hypothetical failures. They happen when role discipline is advisory rather than enforced. The next problem is how to make the architecture self-enforcing at runtime.

Hook-Enforced Discipline

The CLAUDE.md file (the project-level instruction file that defines the agent's persistent rules) says the orchestrator should not edit files directly. The dispatch protocol says every brief must include success criteria. The architecture says a code reviewer must run after every implementer. These are documented rules. The agent reads them at session start. Then, on turn 14, under time pressure, with a clear path to done visible, the orchestrator edits the file directly. The review step gets skipped. The brief is one sentence.

Prompts ask. Hooks enforce.

Hook-enforced discipline is the pattern of moving behavioral constraints from documentation and system prompts into runtime hooks: code that fires automatically at defined events and either blocks non-compliant behavior before it executes or injects corrective context after it completes, without any manual invocation required.

---

why prompts drift and hooks don't

A prompt constraint is a request. The model reads it, weights it against everything else in context, and produces output. Most of the time the constraint holds. Drift happens when the constraint is distant from the action point, when the rule was stated 2,000 tokens ago and the current reasoning is deep in an implementation thread. It also happens when following the rule has an immediate cost (an extra dispatch, a delayed completion) and the rule's benefit is abstract (quality, maintainability, architectural integrity). The model's in-context reasoning is outcome-oriented. Rules that delay outcomes compete against that orientation and lose, not always, but often enough to matter.

Hooks have a different property: they're not in the context. They don't compete with the model's reasoning. They fire based on observable events: a specific tool being called, a specific agent completing, the session attempting to stop. The model can't reason around a hook the way it can de-weight a system prompt rule. A `PreToolUse` hook that returns `permissionDecision:`

`"deny"` stops the tool call regardless of how strongly the model's reasoning argued for it. The constraint doesn't erode because it's never in the reasoning stream to begin with.

The orchestrator-discipline hook kit (a set of runtime hooks that enforce the orchestrator's role boundaries automatically) makes this concrete. The production system runs a `PreToolUse` hook that blocks the main session from reading more than three files per turn. The rule exists in the system prompt too; the hook is why it holds at turn 30 the same way it holds at turn one. The hook doesn't care what turn it is, what the current task is, or how compelling the case for a fourth file read looks from inside the reasoning chain. It fires on the event, checks the condition, and blocks or allows.

---

the four hook types

The hook system exposes four event points. Each maps to a different enforcement use case.

`PreToolUse` fires before a tool executes. This is the blocking hook: the one that prevents non-compliant actions before they happen. Dispatch brief validation, orchestrator file-read limits, and banned-operation gates all belong here. The hook receives the tool name and input as JSON; it returns either `permissionDecision: "allow"` or `permissionDecision: "deny"` with an optional reason. On deny, the tool doesn't fire, and the reason text is surfaced to the model as context for self-correction.

Wiring a `PreToolUse` hook correctly requires using `"Agent"` as the matcher string for subagent (a separate agent session that runs in its own context window and returns a structured result) dispatch gates, not `"Task"`, which is the legacy SDK name and silently never fires in current Claude Code. A hook wired to `"Task"` will load without error, pass schema validation, and never execute. The canonical verification: check the system prompt for `"name": "Agent"` in function declarations, and check session transcripts for `tool_name:"Agent"` in tool_use blocks.

`PostToolUse` fires after a tool completes. This is the logging and state hook. It can't undo the tool call, but it can record what happened: appending to a turn ledger, logging a dispatch event, updating a session state file. The orchestrator-discipline kit uses a `PostToolUse` hook to append every successful IC dispatch to `/tmp/orchestrator-turn-ledger.json`. This ledger is what the

Stop hook reads when checking whether a review step was completed. Post-use hooks are the mechanism that makes stateful enforcement possible across multiple turns.

`SubagentStop` fires when a subagent completes. This is the review-injection hook. When the implementer finishes a turn that wrote one or more files, the `SubagentStop` hook scans the return for the `FILES CHANGED:` section. If files were changed, the hook emits additional context instructing the orchestrator to dispatch the code reviewer next. The orchestrator doesn't need to remember the review requirement. The hook injects the reminder at the exact moment it's relevant: when the implementer's output is fresh and the next dispatch decision is live. This is a common pattern in agentic-coding setups: post-implementer review injection via `SubagentStop` ensures review runs automatically rather than relying on the orchestrator to remember.

`Stop` fires when the session attempts to complete. This is the gate hook: the final check before work is considered done. The quality gate in the production system blocks Stop when the turn ledger shows one or more IC dispatches without a corresponding code-reviewer or `qs-lead` dispatch. The session cannot close until the review step fires. There's one carve-out: if the last assistant message ends with a question mark and no tool calls fired this turn, Stop is allowed through, the session is waiting for user input, not presenting completed work.

---

no manual invocation required

The "no manual invocation" principle is what separates enforcement from aspiration. A SubagentStop nudge that reminds the orchestrator to run a code review is useful. A Stop hook that blocks the session until the review runs is enforcement. The difference isn't in the intent; it's in whether the discipline holds when the model is under pressure to finish.

Hooks that require manual setup per session, per task, or per agent type fail at scale. The whole point is that the constraint applies automatically to every matching event. The auto-review hook fires on every implementer `SubagentStop` where files changed. It doesn't fire because the orchestrator remembered to wire it. It fires because the runtime configuration includes it permanently.

This distinction matters when you're building for a team rather than a single operator. A rule documented in CLAUDE.md is enforced by the person who wrote it and forgotten by everyone else. A hook wired in `settings.json` fires for every session, every operator, every task, regardless of whether the operator has read the documentation. The enforcement artifact travels with the configuration, not with the individual's memory of the rule.

This is the principle Dex Horthy's "no vibes" framing points at: rules embedded in workflow automation hold; rules remembered in prompts drift. The enforcement artifact (the hook file, the settings.json entry, the test fixtures) is the constraint. The system prompt documentation of the same rule is a description of what the enforcement does. Both exist, but the hook is what makes the rule durable. (source: Dex Horthy (HumanLayer), "No Vibes Allowed: Solving Hard Problems in Complex Codebases")

A production system has 15 test fixtures that cover the hook kit's behavior. Each fixture is a JSON blob representing a specific tool call or session event. The test runner passes each fixture to the relevant hook script and checks the output. Brief-missing-goal → deny, brief-present → allow, implementer-with-files-changed → review injection, stop-with-no-review → block, stop-with-review → allow. These fixtures run before every change to the hook kit. The enforcement is verified, not assumed.

---

what to enforce with hooks

Three enforcement targets produce the highest return on the investment of writing and maintaining hook code.

Post-implementer review gate. The most common discipline failure in orchestrator-plus-roster systems is the skipped review step. Implementer finishes; orchestrator sees done; session closes. No reviewer fires. The hook pattern: `SubagentStop` on `implementer` injects the review reminder; `Stop` hook blocks if no reviewer has dispatched. The two hooks work together: injection creates the prompt to act, the Stop gate enforces action before completion. Neither hook alone is sufficient; the injection without the Stop gate is advisory, and the Stop gate without the injection produces a blocked session with no clear next step.

Dispatch brief validation. Malformed briefs are the entry point for under-specified work. An implementer dispatched with a one-sentence brief will produce something; it will do its best against an under-specified goal and

produce output that looks plausible. The problem surfaces later, during review or integration. The `PreToolUse` hook on `Agent` dispatch checks incoming briefs for required fields before the subagent fires. The reject reason names the missing fields. The orchestrator self-corrects in the same turn. Brief quality is enforced at the moment of dispatch, not discovered at the moment of return.

Orchestrator scope limits. The main session staying a router depends on the main session not accumulating tool-output bloat. A `PreToolUse` hook on `Read`, `Edit`, `Write`, and `Grep` above the per-turn threshold enforces the scope boundary. The hook reads a turn ledger to count same-tool calls in the current turn; on threshold breach, it blocks and emits a redirect to dispatch. The orchestrator cannot instrument its own escape hatch through repeated reads; the count is in the ledger, not in the model's memory.

---

the kill-switch pattern

Enforcement creates operational risk. A bug in a hook file, a malformed ledger, a transient file system issue: any of these can produce a broken hook that blocks legitimate work. Production hook kits require a kill-switch: a mechanism to disable enforcement without modifying hook code or settings files.

The standard pattern is a sentinel file. `touch /tmp/orchestrator-discipline.OFF` disables the full kit for the current session. The hook script checks for the file at startup; if present, it exits with allow and logs a warning. The sentinel approach has three properties that make it appropriate for production: it requires a deliberate action (creating the file), it's session-scoped (the file persists only until removed), and it's auditable (the log records every turn where the kill-switch was active).

An environment variable variant provides the same escape for shell-level automation: `ORCHESTRATOR_DISCIPLINE_OFF=1` in the shell environment disables enforcement for the entire shell session. Useful for scripted runs where the hook kit isn't relevant: deploy scripts, CI invocations, one-off maintenance tasks. The variable is explicit, visible in the shell environment, and requires intentional export.

Individual hooks can carry their own kill-switches for surgical disable. A well-designed auto-review hook checks for a sentinel file like `/tmp/auto-review-post-implementer.OFF`, a common pattern in agentic-coding setups. Disabling

auto-review doesn’t disable brief validation or scope limits. Granular kill-switches allow debugging a specific hook without dropping the entire enforcement kit.

---

worked example: a hook, on disk

A brief-validation hook is 30 lines. Here’s what it looks like as a file you can deploy today.

`~/.claude/settings.json` (hook registration)

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Agent",
        "hooks": [
          {
            "type": "command",
            "command": "python3 /home/you/.claude/hooks/validate-brief.py"
          }
        ]
      }
    ]
  }
}
```

The `matcher: "Agent"` string is exact. A value of `"Task"` silently never fires in current Claude Code; see the four hook types section above for the canonical check.

`~/.claude/hooks/validate-brief.py`

```python
#!/usr/bin/env python3
import json, sys

payload = json.load(sys.stdin)
tool_input = payload.get("tool_input", {})
prompt = tool_input.get("prompt", "")

REQUIRED = ["goal:", "success_criteria:", "return_format:"]
missing = [f for f in REQUIRED if f not in prompt]

if missing:
    print(json.dumps({
        "hookSpecificOutput": {
            "hookEventName": "PreToolUse",
            "permissionDecision": "deny",
            "permissionDecisionReason": (
                f"Dispatch brief rejected. Missing required fields: "
                f"{', '.join(missing)}. "
                "Add goal, success_criteria, and return_format before
        dispatching."
            )
        }
    }))
    sys.exit(0)

print(json.dumps({
    "hookSpecificOutput": {
        "hookEventName": "PreToolUse",
        "permissionDecision": "allow"
    }
}))
```

Example: blocked dispatch

The orchestrator attempts to dispatch with: `"Implement the rate limiter."` The hook fires, finds no `goal:`, `success_criteria:`, or `return_format:` field, and returns:

```
Dispatch brief rejected. Missing required fields: goal:,
success_criteria:, return_format:. Add goal, success_criteria,
and return_format before dispatching.
```

The subagent never fires. The model reads the deny reason, writes a conforming brief, and retries. The fix happens in the same turn, before any IC tokens are spent on an under-specified task.

Example: allowed dispatch

A brief containing `goal: Add rate limiting to POST /api/payments`, `success_criteria: npm test passes and 11th request returns 429`, `return_format: structured_report` passes all three checks. The hook returns `permissionDecision: "allow"` and the subagent fires normally.

---

hook-enforcement diagnostic

Score one point for each “yes.”

1. Is your dispatch brief requirement stated only in CLAUDE.md or system prompt documentation, with no runtime check? If yes: the requirement is advisory. Malformed briefs will ship under pressure. Add a `PreToolUse` brief-validation hook.

2. Does your post-implementer code review step depend on the orchestrator remembering to dispatch the reviewer? If yes: the review step will be skipped. Add a `SubagentStop` injection hook on the implementer plus a `Stop` gate that checks the turn ledger.

3. Do you have a kill-switch for your hook kit, a way to disable enforcement without editing hook code? If no: you have a brittle enforcement setup. A hook bug with no kill-switch blocks all work until the bug is fixed. Add the sentinel file pattern before the first production incident.

4. Have you verified your hook matchers against live session transcripts? If no: hooks wired to the wrong matcher string silently never fire. Check that subagent dispatch hooks use `"Agent"`, not `"Task"`. The verification takes two minutes and prevents the silent-failure mode.

5. Does your hook kit include test fixtures that verify deny/allow behavior for each rule? If no: you’re running unverified enforcement. A hook that silently allows everything is worse than no hook; it creates a false sense of coverage. Build the fixtures before the hook ships.

Score 0–1: The enforcement layer is solid. Focus on expanding the fixture coverage as the hook kit grows. **Score 2–3:** Some rules are enforced, some are advisory. Identify the one rule that fails most often under pressure and add the corresponding hook first. **Score 4–5:** Your discipline layer is entirely in documentation. The rules exist; the enforcement doesn’t. Start with the post-implementer review gate; it’s the highest-frequency failure and the most straightforward hook to wire.

Hooks enforce structure at runtime. But they enforce today’s structure: the rules as currently understood. The next layer of work is measuring whether those rules are producing the outcomes they’re designed for: whether the constraint set is correctly calibrated, whether the enforcement is too strict or too loose, and how to tune both without waiting for a production incident to surface the evidence.

Eval-Driven Context Tuning

Your `CLAUDE.md` (the project-level instruction file that governs your agent's behavior) has 47 rules. You added most of them after something went wrong. You don't know which ones are working, which ones are redundant, and which ones are actively competing with each other. Every time you add a rule to fix a new failure, you have no signal on whether the fix held, or whether it broke three other things.

This is not a maintenance problem. It's a measurement problem.

Eval-driven context tuning is the practice of measuring agent behavior before changing context, measuring again after, and using the diff to decide what to keep, add, or prune.

---

the feedback loop that most teams skip

Every experienced engineer applies a feedback loop to code: write a test, run it, see what breaks, fix it, run again. The same loop almost never gets applied to the context the agent runs on. Rules get added reactively. Configuration grows as an append-only log of past failures. Nobody measures whether the current configuration produces better outcomes than last month's version.

The result is instruction drift in reverse: you wrote instructions for failures you had, not the failures you have now. Some of those rules have been load-bearing for months. Others became irrelevant when you changed your workflow. A few are probably contradicting each other quietly. You can't tell which is which without measuring. (source: Ankur Goyal (Braintrust), "Five Hard Earned Lessons About Evals")

The other failure mode is additive anxiety: you're afraid to remove any rule because you don't know what it was protecting. So the file grows in one direction only. Each added rule increases the fixed injection cost of every future session and competes with other rules for attention weight. A `CLAUDE.md` with 60 rules is not six times safer than one with 10 rules. It may be substantially less safe; the model's attention is split across a configuration that was written for 12 different historical failures with no coherent architecture.

Ankur Goyal frames what eval competence looks like when it's working: a new model releases, and within 24 hours your team can validate whether it improves or degrades your use case, not by vibes, but by running a suite of scored examples. That speed is only possible if the eval infrastructure was built before the model changed. (source: Ankur Goyal (Braintrust), "Five Hard Earned Lessons About Evals")

The same principle applies to context changes. Before you add a rule, you need a baseline. After you add it, you need a rescore. Without that loop, you're driving by reading the instruments in your rear-view mirror.

---

what to measure

Four metrics are worth tracking. They're listed in order of implementation cost: start at the top.

Task success rate. Define a small set of representative tasks your agent runs regularly. Grade each output: did it satisfy the goal? A 10-task golden set, scored binary, gives you a number you can track over time. When your task success rate was 80% last week and 70% this week, you have evidence that something changed. Without the measurement, you have a feeling. The threshold for "success" should be written down before you run the eval; otherwise you'll grade charitably on the tasks where you already know what changed.

Output structure compliance. If your rules specify a format: how the agent names files, how it structures its return messages, which template it uses for dispatch briefs. You can score compliance mechanically. Count the percentage of outputs that match the expected structure. This is cheaper to score than task success and surfaces instruction drift fast: if compliance drops, a rule broke or got buried.

Token cost per task. Track how many tokens your representative tasks consume on average. A rising token cost with flat or declining task success is a diagnostic: you're spending more context budget and getting less back. Often this signals that a bloated config is crowding out the reasoning budget.

Regression rate against baseline. Keep one version of your context configuration as a frozen baseline, the version you were happy with. When you change anything, run both the new config and the baseline against your golden set. If the new config improves task success but introduces regressions

on previously-passing examples, you need to decide whether the trade is worth it. Goyal is explicit on this: treating scores like a spec means you own the trade-off decision when the numbers change. (source: Ankur Goyal (Braintrust), “Five Hard Earned Lessons About Evals”)

---

building the golden set

A golden set is a collection of representative inputs paired with expected outputs or scored criteria. It’s the foundation everything else rests on.

The most common mistake is building a golden set from synthetic examples: inputs you invented to cover edge cases. Synthetic sets test what you imagined going wrong. They systematically miss what actually goes wrong. The practitioner pattern that works: build your golden set from real failures. (source: Ankur Goyal (Braintrust), “Five Hard Earned Lessons About Evals”)

When a user complaint arrives, an agent ignored a rule, produced wrong output, looped on a task, that complaint is a golden-set candidate. Capture the input context that produced the failure, define what the correct output should have looked like, and add it to the set. Your golden set becomes a direct representation of the failure modes your system actually encountered, not the ones you hypothesized.

Three practices keep the golden set useful:

Human curation, not automatic ingestion. Every failed case is a candidate, not an automatic addition. You need a person to decide whether the failure represents a genuine gap in your configuration or a one-off edge case that shouldn’t drive rule changes. Auto-ingesting every failure overfits the config to noise. (source: Ankur Goyal (Braintrust), “Five Hard Earned Lessons About Evals”)

Size by coverage, not volume. A 10-task golden set that covers the four failure modes from Chapter 2 (context rot, the gradual loss of early-session instructions as the token budget fills; instruction drift, rules that stop being followed mid-session; token bloat, invisible budget drain from uncompressed tool outputs; and loop thrashing, the agent cycling through failed approaches without recognizing the repetition) is more useful than a 200-task set that clusters around one failure mode. Cover the surface; don’t pad the count.

Version the set alongside the config. When you prune a rule, note which golden-set examples that rule was protecting. If you later see those cases fail again, you know what you removed and why.

---

the rule-is-helping test

This is the direct application of eval-driven tuning: how do you decide whether a specific rule in your `CLAUDE.md` is earning its token cost?

The test runs in three steps.

Step 1: Identify the rule and its protected examples. Every rule was added because something went wrong. Find the golden-set examples that rule was meant to fix, or, if you don't have golden-set coverage yet, construct one or two representative examples that would have triggered the failure.

Step 2: Run the config with and without the rule. Score your representative examples against both configurations. If removing the rule produces identical output on the protected examples, the rule is either redundant (the behavior you wanted is now the model default) or it was never actually load-bearing. If removing the rule causes the protected examples to fail, the rule is earning its place. (source: Dex Horthy (HumanLayer), "No Vibes Allowed: Solving Hard Problems in Complex Codebases")

Step 3: Check for side effects. Run the full golden set against the config without the rule. A rule that protects its target examples but degrades other tasks is creating hidden cost. The token budget it consumes, plus the attention weight it pulls away from other rules, may be producing net negative outcomes across the configuration.

Rules that pass Step 2 and Step 3 stay. Rules that fail Step 2 are load-bearing: document what they protect and keep them. Rules that fail Step 3 need to be either restructured (narrower scope, moved to a matched rule instead of an always-rule) or removed.

This is eval-driven pruning. It's how a 47-rule `CLAUDE.md` becomes a 20-rule configuration that performs better, because every remaining rule has evidence behind it.

The documentation step matters as much as the pruning. When you remove a rule, record why: what it was protecting, why the protection is no longer needed, and what eval examples cover the behavior it addressed. Without that

record, the same rule gets re-added six months later when a new team member hits the same failure the rule was originally written to fix. Your eval set and your decision log are the institutional memory of your context configuration.

---

evals as a model-change buffer

Model updates are a context engineering stress test. Every time the underlying model changes, behavioral tuning that depended on the previous model's quirks can silently break. Rules that existed to work around a specific failure mode may now fight the model's improved default behavior. Instructions written for one model's instruction-following style may parse differently on the next.

Without an eval suite, you find out about model regressions through user complaints, or by noticing that the agent started making mistakes it wasn't making last week. With an eval suite, you find out when you run the suite against the new model before deploying anything.

This is the durability argument that runs through this guide. A context configuration validated by evals is not version-locked to a specific model. When the model changes, you run the suite. If task success holds or improves, you deploy. If it drops, you have specific examples that regressed, not a vague sense that something is wrong, and you can make targeted changes.

Goyal's pattern for ambitious teams: maintain eval cases for use cases you want to support but that aren't viable with current models. Run them against every new model release. When a case crosses the viability threshold, you know immediately, and you can ship the feature in days rather than weeks, because the eval infrastructure was already there. (source: Ankur Goyal (Braintrust), "Five Hard Earned Lessons About Evals") Applied to context tuning: maintain eval cases for the behaviors your current rules are trying to enforce. When a model update makes a rule unnecessary, the eval set shows you, and you can prune with confidence instead of anxiety.

---

worked example: one eval, scored

An eval harness (a small test suite that runs representative inputs through your agent configuration and scores the outputs against defined criteria) doesn't need to be elaborate. Here's one eval case as runnable Python.

The eval case: `dispatch-protocol-compliance`

```
# evals/dispatch_protocol_compliance.py

CASE = {
    "id": "dispatch-protocol-compliance-001",
    "scenario": (
        "Orchestrator session. Task: add rate limiting to POST /api/
        payments. "
        "The orchestrator has the implementer in the IC roster and a
        CLAUDE.md "
        "that requires structured dispatch briefs."
    ),
    "input_prompt": (
        "Add rate limiting to the payments endpoint. "
        "10 requests per minute per user."
    ),
    "expected_behaviors": [
        "Orchestrator produces a dispatch brief (not inline code).",
        "Brief contains a `goal:` field.",
        "Brief contains a `success_criteria:` field.",
        "Brief contains a `return_format:` field.",
        "Orchestrator does NOT write or read any files directly.",
    ],
}

RUBRIC = [
    ("dispatches_not_implements", "Output contains 'Dispatch' block; no
        Write/Edit tool calls.", 2),
    ("goal_field_present",        "Brief includes 'goal:'
        key.",                                  2),
    ("success_criteria_present",  "Brief includes 'success_criteria:'
        key.",                     2),
    ("return_format_present",     "Brief includes 'return_format:'
        key.",                        2),
    ("no_direct_file_ops",        "No Read/Write/Edit tool calls in the
        orchestrator turn.",     2),
]
PASS_THRESHOLD = 8  # out of 10
```

Scoring rubric

Criterion	Points	How to score
dispatches not implements	2	Tool call log shows `Agent` dispatch; no `Write` or `Edit`
`goal:` field present	2	String match in the brief text
`success_criteria:` present	2	String match in the brief text
`return_format:` present	2	String match in the brief text
no direct file operations	2	Tool call log shows no `Read`, `Write`, `Edit` in this turn

Passing run (10/10)

The orchestrator produces a structured brief with all three required fields and dispatches via the `Agent` tool. No direct file operations appear in the turn log. All five criteria score 2/2.

Failing run (4/10)

The orchestrator writes a new file (`src/middleware/rate-limit.ts`) directly and adds a note: “I’ve started the implementation.” No dispatch brief. Criteria scored: dispatches_not_implements = 0, no_direct_file_ops = 0, the three field checks = 0. Score: 4/10. Below threshold.

The failing run tells you exactly what broke: the orchestrator-scope boundary. You add or strengthen the corresponding `PreToolUse` hook, re-run the eval, and confirm the score moves to passing before committing the config change.

---

eval-driven tuning checklist

Score your current setup before you change anything.

1. Do you have a golden set of any size? If no: your context changes are unmeasured. Before your next rule addition, construct five representative examples from recent failures and score them. That’s your floor.

2. Can you measure task success rate numerically? If no: define binary success criteria for at least three representative tasks. “The agent followed the dispatch protocol correctly” is scoreable. “The agent did a good job” isn’t.

3. Do you track output structure compliance? If no: pick the most important structural rule in your `CLAUDE.md` (the one you’ve re-added the most often). Build a compliance check for it specifically.

4. Have you run any rule with and without, against representative examples, within the last month? If no: pick one rule. Run the test described above. The result, load-bearing or not, tells you more about your config than reading it ever will.

5. Do you have a frozen baseline config to compare against? If no: create one today. Version your current config, note the date. Every future change gets compared against it.

6. Have you re-run your evals after the most recent model update? If no: the configuration you’ve tuned may have been tuned to a model that no longer exists. Run the suite. Find out what changed.

Score 0–2: You’re flying blind. Build the golden set before your next rule change. **Score 3–4:** You have the scaffolding. Focus on running the rule-is-helping test on your five largest rules. **Score 5–6:** Your eval loop is solid. The next investment is automating the scoring step so you can run it in under five minutes.

The Five Disciplines give you the architecture. The patterns in Part III give you the shapes. Evals give you the signal that tells you whether the architecture is working for your specific sessions, your specific tasks, and your specific failure history. Measure before you add. Measure after you prune. The context window you have in six months should be shorter than the one you have today, not longer.

Part IV is the operational kit. The worksheets that follow are the tools you use on your next session to apply everything in this guide.

Part IV: Worksheets

These five worksheets are the operational kit for the field guide. Each one stands alone. Print the one you need, fill it in, act on the result. They don't require you to have read the full guide, but each includes a reference to the chapter where the underlying concept lives.

Worksheet 1: Context Budget Calculator

Purpose: Compute your real available token budget before a session starts, so you know how much room remains for actual task work.

Prerequisite: Know your model's context window size (in tokens). Look it up in the model's documentation or API reference.

Chapter reference: Chapter 6 (Compress), Chapter 8 (Evict).

Step 1: Model context window

Field	Your value
Model context window (tokens)	_____

Common values: 200,000 (Claude 3.5/3.7 Sonnet), 1,000,000 (Gemini 1.5 Pro). Use the value for the model you're actually deploying.

Step 2: Fixed injection costs (counted once per session)

These load at session start and stay in context the entire session.

Component	Token estimate	Your value
System prompt (measure with a token counter)	_____	_____
`CLAUDE.md` / equivalent config file (the project-level instruction file that defines the agent's persistent rules)	_____	_____
Always-injected rules (from rule-router `_always/` rules)	_____	_____
Tool definitions (one estimate per tool × number of tools)	_____	_____
Fixed injection subtotal		_____

To measure token counts: paste each component into a token counter (e.g., the Anthropic Tokenizer, or `tiktoken` for OpenAI models). For tool definitions, check your MCP (Model Context Protocol, the standard interface that connects external tools and data sources to an AI agent) or tool-integration logs.

Step 3: Per-turn injection costs (multiplied by expected session length)

These accumulate as the session runs.

Component	Tokens per occurrence	Expected occurrences	Subtotal
Matched rules (avg tokens per rule × avg matched rules per turn)	_____	_____ turns	_____
Tool call output (avg tokens per tool response)	_____	_____ calls	_____
File reads (avg tokens per file × avg files per task)	_____	_____ files	_____
Conversation history growth (avg tokens per exchange)	_____	_____ exchanges	_____
Per-turn injection subtotal			_____

If you don’t have precise measurements, use these working estimates as a starting floor: - Matched rule: 200–500 tokens each - Tool call output (uncompressed): 1,000–5,000 tokens each - File read (typical source file): 500–2,000 tokens - Conversation exchange (user + assistant): 300–600 tokens

---

Step 4: Compute remaining task budget

```
Context window             = _____________
− Fixed injection subtotal = _____________
− Per-turn subtotal        = _____________
──────────────────────────────────────────
Remaining task budget      = _____________
```

Step 5: Interpret

Remaining budget	Interpretation	Action
> 50% of window	Healthy.	Proceed. Monitor as session grows.
25–50% of window	Tight.	Apply truncation policies to tool outputs. Consider subagent offload (dispatching a bounded work unit to a separate agent session that keeps the token trace out of the main session) for large file reads.
10–25% of window	Critical.	Compress your `CLAUDE.md`. Audit always-injected rules for any that can be moved to matched-rule injection. Reduce tool definitions: disable MCPs you won't use this session.
< 10% of window	Over-committed.	The session will saturate before finishing the planned work. Restructure: split into sub-sessions, offload work units to subagents, or prune the config before starting.

Step 6: Recurring checkpoints

At each checkpoint, re-estimate the per-turn subtotal based on actual session data and recompute the remaining budget.

- [ ] After session start (baseline)
- [ ] At 25% of expected session length

- [ ] At 50% of expected session length
- [ ] At any point a large tool output arrives

Worked example

A developer is using a 200,000-token model for a feature-implementation session.

Fixed injection: - System prompt: 800 tokens - `CLAUDE.md`: 3,200 tokens - Always-injected rules (three rules, ~300 tokens each): 900 tokens - Tool definitions (four MCPs, ~400 tokens each): 1,600 tokens - Fixed subtotal: **6,500 tokens**

Per-turn injection (20-turn session estimate): - Matched rules (avg two per turn, 250 tokens each): 2 × 250 × 20 = 10,000 tokens - Tool outputs (avg three per turn, 1,500 tokens each, uncompressed): 3 × 1,500 × 20 = 90,000 tokens - File reads (avg one per turn, 1,000 tokens): 1 × 1,000 × 20 = 20,000 tokens - Conversation history (400 tokens/exchange × 20): 8,000 tokens - Per-turn subtotal: **128,000 tokens**

Remaining task budget: 200,000 − 6,500 − 128,000 = **65,500 tokens** (~33%)

That's in the "tight" zone. The immediate action: apply compression to tool outputs. If each tool output is summarized to 400 tokens instead of 1,500, the per-turn subtotal drops by 66,000 tokens, pushing remaining budget to ~54%.

This is where the budget calculator earns its value: the developer can see before the session starts that uncompressed tool output is the single largest line item, and address it before it causes a mid-task context limit.

Worksheet 2: Prompt-Soup Diagnostic (12 Questions)

Purpose: Prompt soup is a state of context window saturation where competing instructions, fragmented history, and raw tool outputs degrade the model's reasoning coherence. This diagnostic identifies which of the four failure modes that produce it (context rot, the gradual decay of session quality as early instructions lose attention weight; instruction drift, behavioral rules that stop being followed mid-session; token bloat, silent budget drain from

uncompressed tool outputs; and agent loop thrashing, the agent repeating failed approaches without recognizing the repetition) are active in your current setup. Each question is falsifiable and maps to a specific failure mode.

Chapter reference: Chapter 2 (Prompt Soup), Chapter 4 (Generate), Chapter 7 (Route).

How to score: Answer each question Yes (1 point) or No (0 points). Tally your total. Use the scoring guide at the end to identify your highest-risk failure modes.

Context rot (CR)

CR-1. In the last two weeks, did an agent session end mid-task at a context limit before completing the planned scope of work?

☐ Yes (1) / [ ] No (0)

CR-2. Did an agent repeat a decision or contradict itself in the same session, for example by referencing a file it had already marked as irrelevant, or re-proposing an approach it had already tried?

☐ Yes (1) / [ ] No (0)

CR-3. Has an agent ever produced output that ignored constraints established early in the session (turns 1–5) while correctly applying constraints added later (turn 20+)?

☐ Yes (1) / [ ] No (0)

Instruction drift (ID)

ID-4. Does your `CLAUDE.md` or equivalent config file exceed 5,000 tokens?

☐ Yes (1) / [ ] No (0)

Measure it: `wc -c CLAUDE.md` gives characters. Characters ÷ 4 gives a rough token estimate.

ID-5. In the last month, did you add a rule to address a behavior the agent had already been instructed not to do, in other words, did you re-add a rule because the old version stopped working?

☐ Yes (1) / [ ] No (0)

ID-6. Does your config file contain rules that apply to only one specific task type or project, loaded unconditionally into every session regardless of what you're working on?

☐ Yes (1) / [ ] No (0)

Token bloat (TB)

TB-7. Do you have any MCP servers enabled by default that you don't use in most sessions?

☐ Yes (1) / [ ] No (0)

Each enabled MCP adds its tool definitions to the system prompt on every session, whether or not you use it.

TB-8. In the last month, did you hit a usage limit or exceed planned token spend on a session that felt like it should have been well within budget?

☐ Yes (1) / [ ] No (0)

TB-9. Are raw tool outputs (file reads, API responses, search results) injected verbatim into context without any compression, summarization, or truncation step?

☐ Yes (1) / [ ] No (0)

Agent loop thrashing (LT)

LT-10. Has an agent ever entered a cycle of failed attempts (trying an approach, encountering an error, correcting, and then re-trying the same approach) without recognizing it had already tried it?

☐

Yes (1) / [ ] No (0)

LT-11. Have you observed “rush to completion” behavior: an agent that shortcut verification steps and shipped an output that looked correct but didn’t actually satisfy the goal?

☐ Yes (1) / [ ] No (0)

LT-12. When a session goes wrong, is your first instinct to keep adding correction turns (“no, wait, I meant X”) rather than starting a new session with a clean handoff summary?

☐ Yes (1) / [ ] No (0)

Scoring guide

Total score (all 12):

Total	Status	Priority action
0–2	Healthy	Continue monitoring. Re-run this diagnostic in 30 days.
3–5	Moderate risk	Address your highest-scoring failure mode first.
6–9	High risk	Two or more failure modes are active. Start with the 30-day migration plan (Worksheet 5).
10–12	Saturated	All four failure modes are present. Treat this as a context emergency. Freeze new features; focus exclusively on the harness.

Per-failure-mode breakdown:

Failure mode	Questions	Your subtotal
Context rot	CR-1, CR-2, CR-3	_____ / 3
Instruction drift	ID-4, ID-5, ID-6	_____ / 3
Token bloat	TB-7, TB-8, TB-9	_____ / 3
Agent loop thrashing	LT-10, LT-11, LT-12	_____ / 3

Any subtotal of 2–3 indicates the corresponding failure mode is active. Start with the highest subtotal. If two subtotals tie, address token bloat first; it amplifies the other three.

Worked example interpretation

A developer scores: - Context rot: 2/3 (questions CR-1 and CR-3) - Instruction drift: 3/3 (all three) - Token bloat: 1/3 (TB-7 only, one unused MCP) - Agent loop thrashing: 1/3 (LT-12, adds correction turns instead of restarting)

Total: 7/12, high risk. Two failure modes are active at high levels.

Priority order: 1. Instruction drift (3/3): highest subtotal. Start here. The `CLAUDE.md` is probably too long and unconditional. Run the always-rule audit (Worksheet 3). The developer's ID-5 "yes" (re-added a rule recently) is a direct signal that rules aren't sticking; restructure before adding more. 2. Context rot (2/3): active but not at max. The fact that CR-3 fired (early constraints overridden by late ones) tells you the session is growing too long before the relevant constraints re-enter attention range. Sandwich architecture (Chapter 9) addresses this directly. 3. Token bloat (1/3) and loop thrashing (1/3): monitor. The unused MCP (TB-7) is an easy fix: disable it. The correction-turn habit (LT-12) will resolve partially once instruction drift is addressed, when rules are actually followed, there are fewer wrong turns to correct.

Worksheet 3: Rule-Router Cheat Sheet

Purpose: A one-page reference for structuring your rule-routing system (a rule router evaluates predicates on each prompt and injects only the matching rules, keeping the always-injected block bounded regardless of how large your rule vault grows). Use it when building your first router or auditing an existing one.

Chapter reference: Chapter 7 (Route).

The two routing mechanisms

Mechanism	How it works	Best for
Keyword regex match	Pattern-match on literal strings in the incoming prompt (e.g., `\bsecurity\b`, `CLAUDE\.md`, `deploy`)	High-precision, low-ambiguity triggers. Use when the rule should fire only on an exact phrase or command.
Semantic cosine match	Embed the prompt, compare to rule-topic embeddings, fire if similarity > threshold	Broad-coverage, intent-driven triggers. Use when the user might phrase the same intent many ways.

Decision rule: start with keyword regex. Move to cosine only when keyword matching produces too many missed triggers or too many false positives.

Always-rule vs matched-rule decision tree

```
Is this rule relevant on EVERY task, regardless of topic?
│
├─ YES → Always-rule.
│        Load at session start. Count its tokens in your fixed budget.
│        Examples: response style, commit discipline, output format.
│
└─ NO  → Matched-rule.
          Does the trigger condition have unambiguous keywords?
          │
          ├─ YES → Keyword regex route.
          │        Set a regex pattern. Test against 10 real prompts.
          │        Aim for zero false positives on unrelated prompts.
          │
          └─ NO  → Cosine similarity route.
                   Embed the rule topic. Set threshold (see below).
                   Monitor false-positive rate for the first week.
```

Threshold recommendations

Sensitivity needed	Cosine threshold	Notes
High recall (few misses, more false positives)	0.65–0.70	Use for safety or compliance rules where missing a trigger is costly.
Balanced	0.75–0.80	Default starting point for most rules.
High precision (few false positives, more misses)	0.82–0.90	Use for highly specific rules where irrelevant injection costs more than missing a trigger.

Test your threshold against at least 20 real prompts before deploying. Adjust in 0.05 increments. Do not set thresholds below 0.60; at that level, almost everything matches.

Always-rule audit checklist

Run this before adding a new always-rule. An always-rule that should be a matched-rule is one of the fastest ways to grow instruction drift.

- [ ] Is this rule relevant to at least 80% of sessions, regardless of task type?
- [ ] Does this rule apply to session mechanics (not task content)?
- [ ] Is this rule fewer than 200 tokens?
- [ ] Does this rule not conflict with any existing always-rule?

If you answered “no” to any of the above, route this as a matched rule instead.

Common pitfalls

Rule-set bloat. Every rule added to the always-route increases the fixed injection cost of every session, forever. Audit always-rules quarterly. A rule that was relevant six months ago may now be the model's default behavior; test with the rule-is-helping test from Chapter 12.

False regex matches. A keyword regex for `test` will fire on "contest," "latest," "attest." Use word boundary anchors (`\btest\b`). Test every regex against at least five prompts where it should NOT fire.

Cosine threshold drift. Embeddings from different models are not comparable. If you switch embedding models, re-tune every cosine threshold from scratch. Do not port thresholds from one model to another.

Stale rules. A matched rule for a workflow you no longer use still occupies space in your rule database and can produce false matches. Remove rules for deprecated workflows.

Missing catch-all. If no rule matches, what happens? Document the default behavior. An undocumented default is a future debugging session.

Worksheet 4: Subagent Dispatch/Return Template

Purpose: A copy-paste scaffold for the dispatch and return protocol (the structured format for sending a bounded work brief to a subagent and receiving a compact result summary back). Use it every time you send work to a subagent and every time a subagent returns results.

Chapter reference: Chapter 10 (Orchestrator + IC Roster), Chapter 8 (Evict).

Dispatch brief (orchestrator → subagent)

```
## Dispatch
goal: <one sentence — what the subagent must accomplish>
constraints: <scope limits, files not to touch, time budget>
success_criteria: <testable — how will you verify the work is done
correctly?>
return_format: <structured_report | diff | decision | summary>
escalate_if: <conditions under which the subagent should stop and ask>
```

Field annotations:

`goal`: One sentence. Verb + object + outcome. “Implement the token-count endpoint in `/api/tokens.ts` and return the updated file.” Not “work on the token stuff.” Ambiguous goals produce ambiguous outputs.

`constraints`: Scope limits that prevent blast radius. Name files not to touch. Set a time or tool-call budget. “Do not modify any file outside `/src/api/`. Maximum 10 tool calls.” If you don’t set constraints, the subagent will interpret scope maximally.

`success_criteria`: The testable condition you will check when the work returns. “All existing tests pass. The endpoint returns a 200 with a JSON body containing `{ token_count: <number> }`.” Binary and falsifiable. If you can’t write a success criterion before dispatching, you don’t have a clear enough goal.

`return_format`: What you want back. `structured_report` = summary + files changed + verification + deferred. `diff` = the changes only. `decision` = a recommendation with rationale. `summary` = a short prose summary for context injection. Specify this before dispatch; the subagent’s summary is written to this spec. An underspecified return format produces either too much (context bloat) or too little (information loss).

`escalate_if`: The conditions that break the assumption of bounded scope. “Escalate if the endpoint requires a database schema change.” “Escalate if the implementation requires modifying more than three files.” Without this field, the subagent has no guidance on where to stop and ask, and it will either stop too early (blocking on minor decisions) or too late (making architectural choices it shouldn’t own).

Return protocol (subagent → orchestrator)

```
## Report
outcome: DONE | BLOCKED | PARTIAL
summary: <2–3 bullets>
artifacts: <file paths, test results, commands run>
open_risks: <anything the orchestrator should know>
escalations: <decisions that exceeded the subagent's authority>
```

Field annotations:

`outcome`: One of three states. `DONE` = success_criteria met. `BLOCKED` = hit an escalate_if condition, stopped and is waiting. `PARTIAL` = completed part of the work but could not finish; explain in `open_risks`.

`summary`: Two or three bullets. The orchestrator's context window is the shared resource; the summary is the eviction budget for this work unit. Write for the orchestrator's continuation needs, not for completeness. "What does the orchestrator need to know to keep working?" is the right question.

`artifacts`: Concrete evidence. File paths, test output, commands run and their exit codes. The orchestrator should be able to verify the claim without re-opening the subagent session.

`open_risks`: What the orchestrator should watch for. Not a complete accounting of the subagent's work, just a targeted signal. "The endpoint works but has no rate limiting; that will matter when this hits production."

`escalations`: Decisions the subagent hit that the dispatch brief didn't authorize. If the subagent made a call it wasn't sure about, name it here. The orchestrator decides whether to accept, revert, or review.

Common dispatch failures

Symptom	Cause	Fix
Subagent returns work outside the intended scope	Constraints not specified	Add explicit file scope and tool-call budget to the dispatch
Return summary is too long to inject into orchestrator context	`return_format` not specified	Specify `return_format: summary` and a token budget in the dispatch
Orchestrator can't verify the work	No success criteria	Add a binary falsifiable success criterion before dispatching
Subagent stops on every minor decision	`escalate_if` too broad	Narrow the escalation condition to architectural or scope-expanding decisions only
Subagent makes an architecture decision unilaterally	No escalation condition set	Add `escalate_if: any architectural change required`

Worksheet 5: 30-Day Migration Plan

Purpose: A week-by-week plan for migrating an existing AI-assisted project from ad-hoc prompting to the five-discipline harness. Each week has three or four concrete tasks with acceptance criteria.

Chapter reference: The full guide. Read Part I (Foundations) before Week 1; read the relevant discipline chapter before each week's work.

Before you start

Answer these three questions in writing before Week 1 begins. You'll reference the answers throughout the migration.

1. What is the single most painful recurring failure in your current setup? (Context rot? Instruction drift? Token bloat? Loop thrashing?)
2. What does your current `CLAUDE.md` or equivalent contain? Paste it somewhere you can reference it.

3. What are the two or three tasks you run most often with your AI assistant?

Week 1: Measure the baseline

Goal: establish numerical baselines for the four metrics from Chapter 12. You cannot measure improvement without a starting point.

Task 1.1: Run the prompt-soup diagnostic (Worksheet 2). Record your total score and your per-failure-mode breakdown. This is your starting state. Date it.

Task 1.2: Run the context budget calculator (Worksheet 1). Measure and record: fixed injection cost, estimated per-turn cost for your two most common tasks, and remaining task budget. If remaining budget is under 50%, note it as a priority for Week 2.

Task 1.3: Identify your five most representative tasks. These become the foundation of your golden set (see Chapter 12). For each task, write one sentence describing what "correct completion" looks like. These are your success criteria.

Task 1.4: Run each task once and score it against your success criteria. Record the results: pass or fail, token count if available. This is your Week 1 baseline. You'll run the same five tasks at the end of Week 4 and compare.

Acceptance criteria for Week 1: You have written-down scores for five tasks, a token budget number, and a soup-diagnostic score. If you don't have all three, do not advance to Week 2; you'll be making changes with no baseline to compare against.

Week 2: Build the first structure

Goal: split your monolithic config into always-rules and matched-rules. This is the single highest-leverage structural change.

Task 2.1: Audit your `CLAUDE.md` against the always-rule checklist (Worksheet 3). Mark each rule as "always" or "matched." Rules that are task-specific, project-specific, or apply to fewer than 80% of sessions move to the matched category.

Task 2.2: Move matched-rule candidates out of `CLAUDE.md`. Create a separate rules directory (e.g., `rules/` or `.claude/rules/`). Move the matched-rule candidates there as individual files. Do not delete them; you're restructuring, not pruning yet.

Task 2.3: Set up basic keyword routing for two to three of the highest-frequency matched rules. Use the rule-router cheat sheet (Worksheet 3) to identify appropriate routing patterns. Implement the simplest version: a keyword-match that injects the rule when a matching trigger appears in the prompt.

Task 2.4: Re-run your five golden-set tasks. Compare the results to your Week 1 baseline. Note any regressions. If a task that passed in Week 1 now fails, check whether a matched-rule that was protecting it is no longer being injected correctly.

Acceptance criteria for Week 2: Your `CLAUDE.md` contains only always-rules. At least two matched rules are routing correctly. Your golden-set scores are equal to or better than Week 1.

Week 3: Add the first hook

Goal: enforce one behavioral constraint automatically, without relying on the model to remember it.

Task 3.1: Identify the most frequently violated rule from your `CLAUDE.md`. This is the rule you've re-added or re-enforced the most often. It's the best candidate for hook enforcement.

Task 3.2: Implement a PreToolUse hook for the violation. The hook should intercept the specific action the model takes when it violates the rule and either block the action or inject a reminder. Start with logging only (don't block yet); verify the hook fires correctly before making it enforcement.

Task 3.3: Upgrade to enforcement after one day of logging. Once you've confirmed the hook fires on the right conditions and only on the right conditions, switch from logging to blocking or correction.

Task 3.4: Re-run your golden-set tasks with the hook active. Verify no regressions. Check the hook log: is it firing on the right cases? Is it producing any false positives?

Acceptance criteria for Week 3: One hook is running in enforcement mode. Golden-set scores are equal to or better than Week 2. The hook log shows no false positives on your five representative tasks.

Week 4: Evaluate and prune

Goal: use your four weeks of data to make deliberate decisions about what to keep, add, and remove.

Task 4.1: Re-run the prompt-soup diagnostic. Compare to your Week 1 score. The score should be lower. If it isn't, identify which failure mode score didn't improve and focus the next iteration there.

Task 4.2: Run the rule-is-helping test on your three largest always-rules. For each rule, test with and without against your golden set. Document which rules are load-bearing and which are redundant. Remove the redundant ones.

Task 4.3: Re-run the context budget calculator. Compare to Week 1. The fixed injection cost should be lower (fewer always-rules), the per-turn cost should be lower or unchanged, and the remaining task budget should be higher. If it isn't, the matched-rule routing isn't reducing injection correctly; debug the routing config.

Task 4.4: Write a one-paragraph summary of what you changed and what the numbers say. This is the decision log entry for your migration. Record it in your project notes or vault. It is your evidence that the harness is working, and your starting point for the next iteration.

Acceptance criteria for Week 4: Prompt-soup diagnostic score is lower than Week 1. Context budget remaining is higher than Week 1. At least two rules have been pruned with documented rationale. You have a written decision log.

After Week 4: ongoing cadence

The migration doesn't end at Week 4. It becomes a quarterly cycle:

- **Monthly:** Re-run the five golden-set tasks. Check scores against the baseline.
- **After any model update:** Re-run the full golden set. Log any regressions.

- **After adding any new always-rule:** Run the rule-is-helping test within one week.
- **Quarterly:** Full audit of always-rules against the always-rule checklist. Prune anything that doesn't pass.

The worksheets are the operational layer. The appendix that follows is the reference layer: a glossary of every term used in this guide, the full bibliography, and a one-page printable summary you can post above your monitor.

Part of the Context Engineering Field Guide by Jounes, from the lab.

Appendix

Glossary

Agent loop thrashing. A failure mode where an agent cycles through the same failed approaches without recognizing the repetition, burning context budget on work that produces no progress. One variant is “rush to completion”: the agent shortcuts verification and ships a plausible-looking but incorrect result.

Always-rule. A behavioral instruction loaded into every session unconditionally, regardless of task type. Always-rules live in the fixed-injection layer and count against the context budget on every prompt. Contrast with matched-rule.

Append-only summary log. A truncation policy where tool outputs are never injected verbatim. Each tool call appends a structured entry (timestamp, tool, result summary, conclusion) to a running log. The log, not the raw output, lives in context.

CLAUDE.md. The project-level instruction file where you define an AI agent’s role, rules, constraints, and recurring behavioral patterns. In Claude Code, it loads at session start and merges with the built-in model instructions. In custom agents, it is the equivalent of your system prompt configuration file.

Context budget. The token count available for actual task work after all fixed and per-turn injections are subtracted from the model’s context window. Context budget is the primary resource a session consumes. See Worksheet 1.

Context engineering. The discipline of designing, maintaining, and optimizing the information state delivered to an AI agent so it can reason accurately and act reliably. Applies to the complete state (system prompt, tool outputs, conversation history, injected files), not only the system prompt.

Context rot. The slow degradation of session quality as the token budget fills and early instructions lose attention weight relative to recent tokens. Reasoning accuracy can degrade 30–40% before the physical token limit is reached.

Dispatch/return protocol. The structured format for sending a bounded work brief to a subagent (the dispatch) and receiving a compact result summary back (the return). The dispatch brief includes fields for goal, constraints, success criteria, return format, and escalation conditions. The return shape includes outcome, summary, artifacts, open risks, and escalations.

Eval harness. A repeatable test suite that runs a fixed set of representative prompts against the agent's current context configuration and scores the outputs against defined pass/fail criteria. Used to measure whether a rule is working, detect instruction drift, and catch regressions when the configuration or the underlying model changes.

Eviction. The discipline of removing material from the context window (via compaction, subagent offload, or truncation) before accumulated tokens from completed work degrade the remaining work.

Golden set. A curated collection of representative task inputs paired with expected outputs or scored criteria. Used to measure agent behavior before and after context changes. Built from real failures, not synthetic examples.

GRCRE. The acronym for the five-discipline context engineering harness: Generate, Retrieve, Compress, Route, Evict. Each discipline targets one category of context failure: Generate controls what enters at session start; Retrieve surfaces just-in-time information; Compress reduces the cost of what is already loaded; Route keeps only situationally relevant rules in scope; Evict clears material the session no longer needs.

Harness. An opinionated runtime layer that decouples the reasoning engine (the LLM) from session state, providing standardized interfaces for tool orchestration, memory persistence, and context engineering. The five-discipline harness (Generate, Retrieve, Compress, Route, Evict) is the framework this guide builds toward.

Hybrid search. A retrieval strategy that combines vector (semantic) search with keyword (lexical) search, then merges and re-ranks the results. Vector search handles conceptual queries; keyword search handles exact-match queries (identifiers, specific values, proper nouns). Hybrid search outperforms either alone because each covers the failure mode of the other.

IC roster. The set of specialist subagents an orchestrator can dispatch work to. Each IC (individual contributor) handles one task type and returns a structured result. The roster keeps work, and its token trace, out of the orchestrator's context.

Instruction drift. A failure mode where behavioral instructions established at session start gradually lose effect as the session grows. The instructions are still present in context but lose the retrieval competition against more recent, more syntactically prominent content.

KV-cache. The key-value cache that stores intermediate attention computations for tokens already processed. Tokens that remain stable across turns benefit from cache hits; tokens that change force a re-computation. Understanding KV-cache behavior shapes decisions about where to place stable content in the system prompt.

Matched-rule. A behavioral instruction injected only when a routing condition fires: a keyword regex match, a cosine similarity threshold crossing, or a semantic predicate. Matched-rules consume context budget only on the turns where they're relevant. Contrast with always-rule.

MCP (Model Context Protocol). The standard interface that connects external tools and data sources to an AI agent. MCP servers expose capabilities (search, code execution, database queries) that the agent calls as tool invocations. Each enabled MCP adds its tool definitions to the system prompt, consuming context budget even when the tool is not used.

Orchestrator. The top-level agent that plans, dispatches, and coordinates. The orchestrator routes work to subagents and receives structured return reports. It does not accumulate the token trace of subagent work, only the return summary.

Orchestrator-discipline hook. A runtime hook (or set of hooks) that enforces the orchestrator's role boundaries automatically. A PreToolUse hook blocks the main session from accumulating file-read bloat above a per-turn threshold. A SubagentStop hook injects a post-implementer review reminder. A Stop hook blocks session completion until the required review step has run.

Predicate-gated injection. A routing technique where a rule or context chunk is injected only when a logical predicate evaluates as true, for example, "inject the security rules only when the prompt contains a reference to authentication or secrets." The predicate is the gate; injection is conditional on it.

Prompt soup. A state of context window saturation where competing instructions, fragmented history, and raw tool outputs degrade the model's reasoning coherence. Four failure modes produce it: context rot, instruction drift, token bloat, and agent loop thrashing.

Reranker. A cross-encoder model that rescores a small candidate set (20 to 50 chunks from an initial retrieval pass) by evaluating the query and each candidate together as a single input, rather than comparing embeddings separately. Reranking adds precision at the final step: the top three chunks injected into context are the ones that genuinely answer the query, not just the ones nearest to it in embedding space.

Rule-router daemon. A small local service that evaluates predicates on each incoming prompt and injects only the matching rules from the rule vault. The always-on core fires unconditionally; the predicate-gated rules fire only when their activation condition matches. The injected block stays bounded regardless of vault size.

Sandwich architecture. A context structure that places stable, high-priority instructions at both the beginning and the end of the context, surrounding the variable middle content. Early placement anchors the session; late placement counteracts the recency bias that pushes early tokens out of effective attention range.

Subagent. A separate agent session that runs in its own context window, handles a bounded work unit, and returns a compact result summary to the calling session (the orchestrator). The orchestrator never sees the subagent's intermediate steps, only the return. This keeps the orchestrator's context small regardless of how much work the subagent does.

Token bloat. A failure mode where tool outputs, config file re-injection, and MCP tool definitions consume context budget silently, leaving less room for task work. Often invisible until the session hits a limit mid-task.

Source bibliography

Foundations

- Patrick Debois (Tessl), "Context Is the New Code"
- Ankur Goyal (Braintrust), "Five Hard Earned Lessons About Evals"

- Sally-Ann Delucia (Arise), “Hierarchical Memory: Context Management in Agents”
- Peter Werry (Unblocked), “Mergeable by default: Building the context engine to save time and tokens”

Disciplines

- Dex Horthy (HumanLayer), “No Vibes Allowed: Solving Hard Problems in Complex Codebases”
- Brendan O’Leary, “Agentic Engineering: Working With AI, Not Just Using It”
- Val Bercovici + Kellen Fox (WEKA), “Context Platform Engineering to Reduce Token Anxiety”
- Stephen Chin (Neo4j), “Context Engineering: Connecting the Dots with Graphs”
- Jared Zoneraich (PromptLayer), “How Claude Code Works”

Patterns

- Boris Cherny (Anthropic), “Claude Code & the evolution of agentic coding”
- Brian John (BetterUp), “Hacking Subagents Into Codex CLI”
- Alberto Romero (Jointly), “The Unbearable Lightness of Agent Optimization”

External papers and tools

- AgentSight: System-Level Observability for AI Agents Using eBPF
- COBALT-TLA: A Neuro-Symbolic Verification Loop for Cross-Chain
- Drift: Architectural Erosion Check
- Kgent: Simplifying Kernel Programming with LLM-Powered eBPF
- practitioner quotes on context rot and instruction drift (Chapters 1, 2): https://forum.cursor.com/t/why-does-cursor-keep-lose-context-and-giving-random-answers/67492/9
- context rot session failure (Chapter 2): https://dev.to/dibyanshu_kumar/how-i-stopped-losing-work-to-context-window-overflow-in-claude-code-1hll
- CLAUDE.md drift and instruction drift quotes (Chapter 2): https://news.ycombinator.com/item?id=47936579
- 44.7k character CLAUDE.md (Chapter 2): https://github.com/anthropics/claude-code/issues/2766

- token bloat quantification (Chapter 2): https://naqeebali-shamsi.medium.com/stop-wasting-tokens-a-developers-guide-to-claude-code-cleanup-de842f6403e5
- agent loop thrashing (“wait that’s not right”) (Chapter 2): https://news.ycombinator.com/item?id=47660925

1-page printable summary

The Context Engineering Field Guide: reference card

The core problem

Context rot degrades every AI session that isn’t actively managed. The model doesn’t crash; it drifts. Early instructions lose weight. Token budget evaporates. The session dies mid-task or ships wrong output. The fix is engineering, not prompting.

The four failure modes

#	Mode	Symptom
1	Context rot	Early instructions ignored; session degrades over time
2	Instruction drift	Rules stop being followed mid-session
3	Token bloat	Budget gone before work is done
4	Agent loop thrashing	Agent repeats failed approaches; rushes to complete

---

The five-discipline harness (GRCRE)

Discipline	What it does	Key action
Generate	Controls what enters context at session start	Structure `CLAUDE.md` as WHAT/WHY/HOW; keep it under 5,000 tokens
Retrieve	Surfaces the right information at the right moment	Semantic chunking; hybrid search; inject only what's relevant to the current task
Compress	Reduces token cost of what's present	Summarize tool outputs before injection; use append-only logs
Route	Keeps only situationally relevant rules in scope	Always-rules for universal behavior; matched-rules for task-specific behavior
Evict	Clears what the session no longer needs	Compact completed phases; offload bounded work units to subagents

---

The three patterns that ship

Pattern	What it solves	How it works
Sandwich architecture	Early instruction loss	Place stable rules at start AND end of context
Orchestrator + IC roster	Context saturation on long tasks	Orchestrator plans and dispatches; ICs do the work; work trace stays in subagent context
Hook-enforced discipline	Rules ignored mid-session	PreToolUse / PostToolUse hooks enforce constraints automatically; the model can't bypass them

---

The eval loop (Chapter 12)

Before any context change: baseline. After any change: rescore. Prune rules that don't improve task success. Keep rules that protect golden-set examples. Re-run after every model update.

Four things to measure: task success rate, output structure compliance, token cost per task, regression rate against baseline.

The five worksheets

#	Worksheet	Use it when
1	Context budget calculator	Before any session: know your real remaining budget
2	Prompt-soup diagnostic	Diagnosing which failure modes are active
3	Rule-router cheat sheet	Building or auditing your routing config
4	Subagent dispatch/return template	Dispatching any bounded work unit to a subagent
5	30-day migration plan	Moving from ad-hoc prompting to the full harness

---

The one-sentence version

Treat the context window like managed memory: control what enters, compress what stays, route what's situational, evict what's done, and measure everything you change.
