Day 6: Multi-Agent Context Management

All About AGENTS.md — Compression, Isolation, Sub-Agent Architecture, and the Central Brain Pattern

Isham Rashik · AI Engineer
Engineering AI with Clarity
NLP • Computer Vision • Fine-Tuning • RAG • Agentic AI

Version: v1.0 · April 25, 2026

Acknowledgment

I am deeply grateful to Dr. Sreedath Panat of **Vizuara Technologies** for creating and generously sharing the *Context Engineering* course. These notes would not exist without their exceptional teaching, clear explanations, and dedication to making cutting-edge AI concepts accessible to everyone.

This session completes the WSCI framework by covering the final two operations — Compress and Isolate — building directly on the Write and Select operations from previous sessions.

→ LLM Context Engineering Bootcamp

Contents

1 Prerequisites

- Understanding of context windows, context rot, and the six core elements (Day 1)
- Familiarity with `CLAUDE.md`, system prompts, persistent memory, and scratch pads (Days 2–3)
- Understanding of the WSCI framework: WRITE, SELECT, COMPRESS, ISOLATE [1] (Day 3)
- Knowledge of MCP architecture: client, server, three primitives, transport layers (Day 4)
- Understanding of tool schemas, tool results, and their impact on the context window (Day 4)
- Understanding of agents: Tool Calling Agent, ReAct Agent, and how agents make API calls (Day 4)

2 Overview

This guide covers **Compress** and **Isolate** — the last two operations in the WSCI framework [1], completing the four-part toolkit introduced on Day 3. Where WRITE creates persistent memory and SELECT retrieves external knowledge via RAG and MCP, Compress and Isolate address what happens *inside* the context window as it fills up during extended agent sessions.

The guide opens with the **Central Brain pattern** — a multi-agent orchestration architecture where specialized agents (each a separate, independent project) are connected through a central orchestrator via two-way MCP. This pattern is the destination the rest of the guide builds toward.

With the big picture in place, the compression chapters cover the two reasons compression exists — cost and context rot — then walk through four techniques (LLM-based summarization, tool result clearing, priority-based trimming, and hierarchical semantic compression) with concrete metrics for compression ratio and the **Information Retention Ratio (IRR)**, the primary measure of compression quality.

The isolation chapters then cover the sub-agent architecture (separate API calls and context windows that prevent any single agent from drowning in irrelevant tokens), the `AGENTS.md` folder hierarchy, the distinction between `CLAUDE.md` and `AGENTS.md`, and five multi-agent patterns — contract-first design, fan-out/fan-in, full isolation, shared base context, and sequential pipeline. A **BiteBridge** food-delivery incident-response simulation grounds every concept in concrete token counts.

3 From Single Agents to Multi-Agent Systems

Day 4 introduced two agent architectures: the **Tool Calling Agent** (implicit reasoning, parallel tool execution) and the **ReAct Agent** (explicit Thought–Action–Observation loops). Both are powerful, but both operate as *single agents* with a *single context window*. Every tool result, every reasoning step, every intermediate output accumulates in that one window until context rot degrades the output quality.

This limitation becomes acute in real-world projects. A software team does not assign one person to write code, run tests, review security, deploy to production, and update documentation. It assigns specialists. The same principle applies to agents: when a project demands research, coding, testing, and review, packing it all into one 200K-token context is the same anti-pattern at the architectural level — one generalist trying to cover every specialty.

Figure 1: From a solo agent juggling everything to a coordinated team of specialists: multi-agent systems divide and conquer

This session addresses two questions that arise naturally from Day 4's agent foundations:

1. **Compress:** When a single agent's context window fills up during extended operation, how do we shrink it without losing critical information? What gets removed first, what is sacred, and how do we measure the quality of compression?
2. **Isolate:** When a task is too large for one context window, how do we distribute it across multiple agents with separate context windows? How does the orchestrator decide what context each sub-agent receives? How do sub-agents communicate without sharing memory?

Together with WRITE and SELECT from earlier days, these two operations close out the WSCI framework: COMPRESS keeps a single window clean, ISOLATE spreads work across many.

Before unpacking either mechanism, the next chapter introduces the **Central Brain pattern** — the architecture that ties them together in production systems.

4 The Central Brain Pattern

The **Central Brain pattern** is a multi-agent architecture in which several specialized agents — each one a **separate, independent project with its own codebase** — are wired together through a single orchestrator over **two-way MCP** [2]. Imagine a research agent, an email agent, a code-review agent, and a data-analysis agent, each living in its own repository, each excellent at one thing. The Central Brain is the project that knows about all of them, decides *which* agent should act *when*, and routes work between them.

This is a sharper kind of multi-agent system than the sub-agent setups Claude Code spawns inside a single project. There, the orchestrator and its sub-agents share a parent process and a coding context. In the Central Brain pattern, every agent is fully independent: separate process, separate memory, separate codebase — communicating only through MCP messages.

We open the guide with this pattern for two reasons. First, it is the *destination* that gives every other technique its purpose: compression keeps each agent's window clean, isolation keeps the agents from contaminating each other, and `AGENTS.md` files configure them. Without a target architecture in mind, those techniques look like a grab-bag of tricks. Second, the pattern was previewed in Day 3 (when discussing persistent memory lifetimes across agent networks) and depends on MCP as its communication layer (Day 4) — so the prerequisites are already in place.

4.1 The Problem: Isolated Agents Across Projects

In a real-world organization, multiple specialized agents may exist as separate projects:

- **Agent 1 (Research):** Searches the web, analyzes papers, compiles findings
- **Agent 2 (Email):** Drafts and sends emails, manages inbox
- **Agent 3 (Code Review):** Reviews pull requests, checks code quality
- **Agent 4 (Data Analysis):** Queries databases, generates reports

Each agent excels at its specialty but has **no awareness of the other agents or the overall company workflow**. Agent 1 does not know when Agent 2 should send a follow-up email. Agent 3 does not know that Agent 4's report is a prerequisite for a code change.

> ⚠ **Warning**
>
> Without orchestration, each agent operates in isolation. The user becomes the "human API," manually deciding which agent to invoke, in what order, and what context to pass between them. This is the same problem that MCP solved for tools: manual coordination does not scale.

4.2 Architecture: Central Brain as Orchestrator

The solution is a fifth project: a **Central Brain** that acts as the orchestrator via two-way MCP.

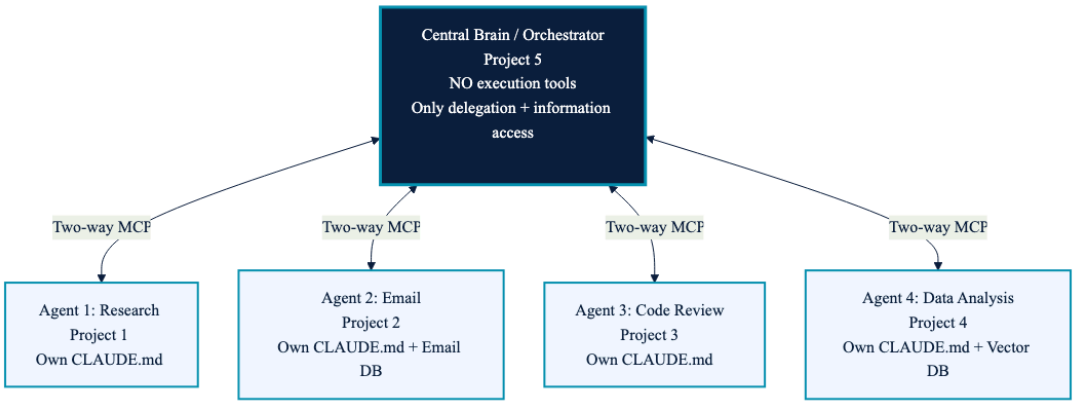

Figure 2: The Central Brain pattern: five separate projects connected via two-way MCP. The orchestrator delegates but never executes.

Five separate projects exist: four specialized agents and one Central Brain orchestrator. The Central Brain connects to each agent via two-way MCP, meaning communication flows in both directions. The Central Brain has **no execution tools of its own**. It cannot research, email, review code, or analyze data. Its only capabilities are (1) accessing information about each agent's status and capabilities, and (2) delegating tasks to the appropriate agent.

4.3 Role Separation: Knows *When* and *What*, Not *How*

The Central Brain holds the *global picture*: what projects are in progress, what tasks are pending, what dependencies exist. Individual agents are focused specialists. The Central Brain provides *instructions*; individual agents provide *execution*. This separation mirrors the MCP client-server split: the orchestrator (like the client) decides *what* to do; the agents (like servers) decide *how* to do it.

Table 1: Central Brain vs. specialized agents: role separation across knowledge, tools, decisions, context, and protocol

Role	Central Brain (Orchestrator)	Specialized Agents
Knows	When tasks are needed, what to delegate, in what order	How to execute a specific assigned task
Has tools for	Information access, delegation, scheduling	Domain-specific execution (email, research, code, data)
Makes decisions about	Task ordering, agent selection, workflow dependencies	How to accomplish a specific assigned task
Context includes	CLAUDE.md files from all agents, global task board, company priorities	Only its own CLAUDE.md, domain-specific databases, current assignment
MCP role	Acts as MCP client to each agent's MCP server	Acts as MCP server exposing its tools to the Central Brain

> 💡 **Idea**
>
> Technically, agents *can* be connected directly via two-way MCP without a central orchestrator. But then no single entity holds the global workflow context — each agent would need to understand the full company picture to know when to delegate to another agent, defeating the purpose of specialization. The Central Brain exists precisely so that specialists can *stay* specialized.

4.4 Shared Context Across the Agent Network

Certain files are referenced across the entire multi-agent network:

- **`CLAUDE.md` files from each agent:** Referenced by the Central Brain to understand each agent's core philosophy, capabilities, and implementation strategy. These are persistent memory (weeks/months lifespan).
- **Shared databases:** e.g., an email agent's vector database may be queried by both the email agent and the Central Brain for different purposes.
- **Task state:** The Central Brain maintains a global understanding of which tasks are assigned, pending, or completed across all agents.

> **Memorize**
>
> The `CLAUDE.md` file is the contract between an agent and the Central Brain. It tells the orchestrator: here is what I can do, here are my constraints, and here is my operating philosophy. The Central Brain reads these files to make informed delegation decisions, never needing to understand the agent's implementation details.

4.5 Central Brain and MCP: The Connection

The Central Brain pattern maps directly onto every MCP concept covered in Day 4:

Table 2: Every MCP concept maps to the Central Brain pattern

MCP Concept	Central Brain Application
MCP Client	Central Brain acts as client to each agent
MCP Server	Each specialized agent acts as a server exposing its tools
Three Primitives	Agents expose tools (execute tasks), resources (CLAUDE.md, databases), and prompts (standard task templates)
Two-way MCP	Bidirectional: Central Brain delegates to agents; agents report status back
Transport	stdio if all agents are on the same machine; HTTP/SSE if distributed
Handshake	Central Brain discovers each agent's capabilities during initialization
Tool selection	Central Brain's LLM decides which agent to delegate to: the same DECIDE step from the tool life cycle

4.6 In Practice: Two Real-World Central Brains

The instructor demonstrated this pattern with two production examples — one internal to Vizuara, one personal to the instructor.

Example 1: Vizuara's Central Brain. An agent orchestrator connected to multiple specialized sub-agents (including an email agent and a segregator agent) via two-way MCP. Each agent follows a shared schema defined by the central orchestrator — the contract-first pattern applied at the inter-project level. This is the canonical Central Brain deployment: many specialists, one coordinator, all stitched together by MCP.

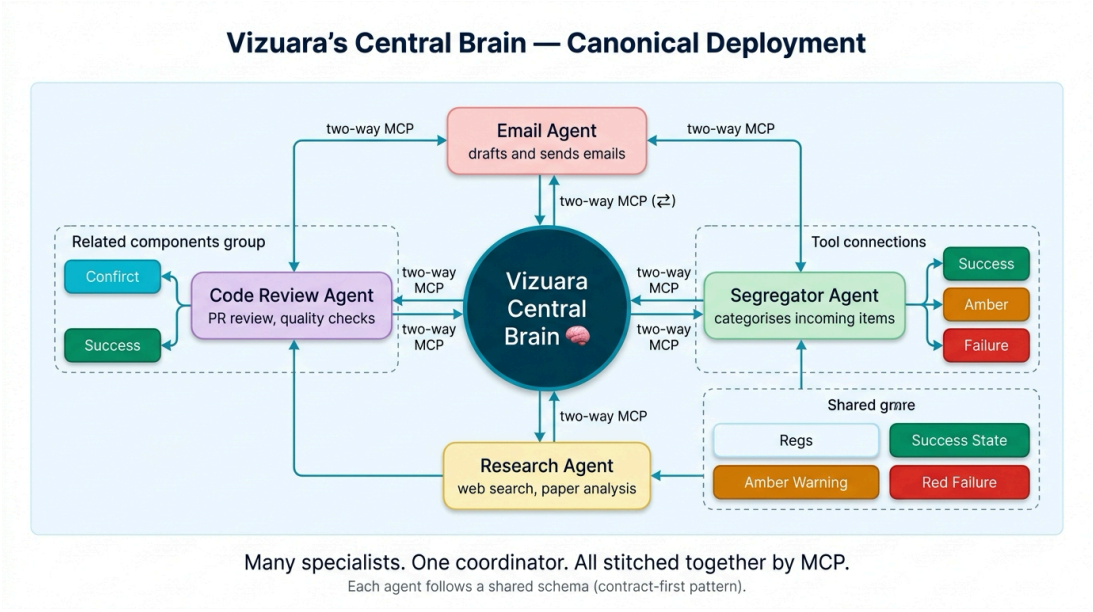

Figure 3: Vizuara's Central Brain: four specialists, one coordinator, all wired through two-way MCP

Example 2: OpenClaw orchestrating Claude Code. [3] **OpenClaw** — a personal AI assistant framework built by Pete Steinberger — acts as a Central Brain orchestrating Claude Code itself. OpenClaw follows many of the same compression and isolation principles discussed in this guide, but its role is purely orchestration: it *delegates* to Claude Code and *never executes* code itself.

Setup and security isolation. OpenClaw runs on an **EC2 instance** (not a personal machine) with no access to personal data — no calendar, no Gmail, no Slack. The only sensitive asset exposed is the API key. The sole communication channel is **Telegram**, chosen because it was never used for personal messaging — it functions purely as a command interface.

The orchestration loop:

1. The instructor sends a message to OpenClaw via Telegram — as naturally as texting a personal assistant.
2. OpenClaw (running Gemini 2.5 Flash) plans the task and delegates execution to Claude Code (running Opus 4.6).
3. Claude Code executes the heavy work — writing code, running tests, managing its own context window with compaction.
4. If Claude Code's session token limit runs out, OpenClaw detects the stoppage and **automatically switches the account**.
5. If any issue arises, OpenClaw **notifies the instructor via Telegram** — even if the laptop is closed and the instructor is away from their desk.

This creates a fully remote, asynchronous development workflow: everything runs on EC2, the instructor can close their laptop and go home, and any issues surface as phone notifications.

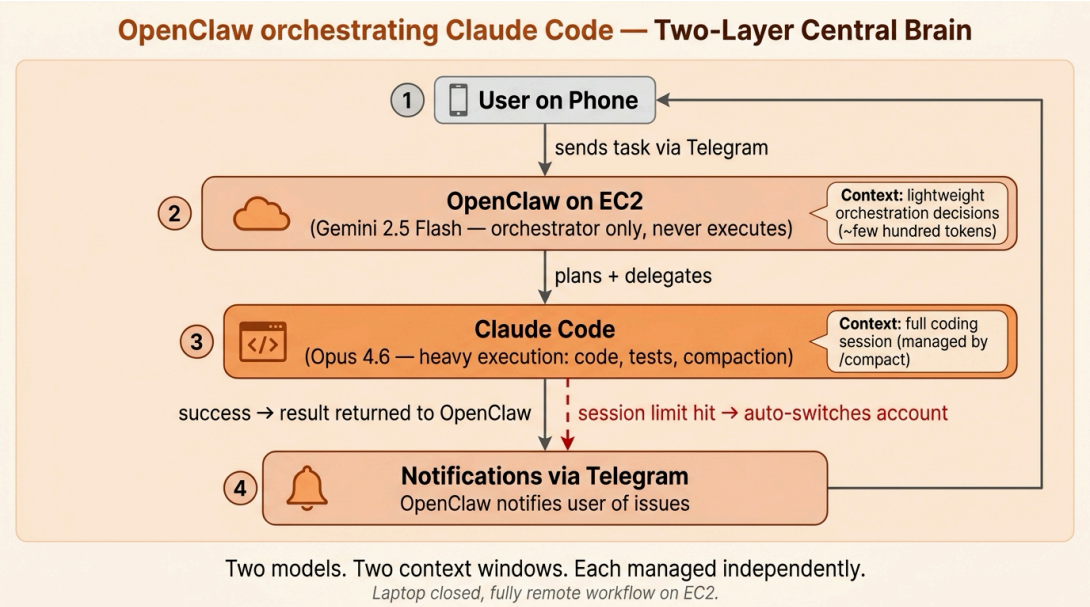

Figure 4: The OpenClaw orchestration loop: two layers, two models, two context windows — each managed independently

> **Warning**
>
> OpenClaw's token consumption can be extremely high with expensive models. The instructor reported $7–8 for 10 minutes with a preview model, but switching to Gemini Flash reduced costs by 100–150×. Always use the cheapest viable model for orchestration layers.

OpenClaw does not suffer from context rot precisely *because* of the Central Brain principle: its Gemini context contains only lightweight orchestration decisions, while all heavy context (code, test results, error traces) lives in Opus 4.6's window, managed by Claude Code's own compaction system. Two layers, two models, two context windows — each managed independently.

With this architectural destination in mind, the remaining sections cover the specific techniques — compression, isolation, `AGENTS.md`, and multi-agent patterns — that make the Central Brain pattern possible.

5 Why Compress: Cost and Context Rot

There are two fundamental reasons to compress the context window. Both are direct consequences of how LLM pricing and attention mechanisms work.

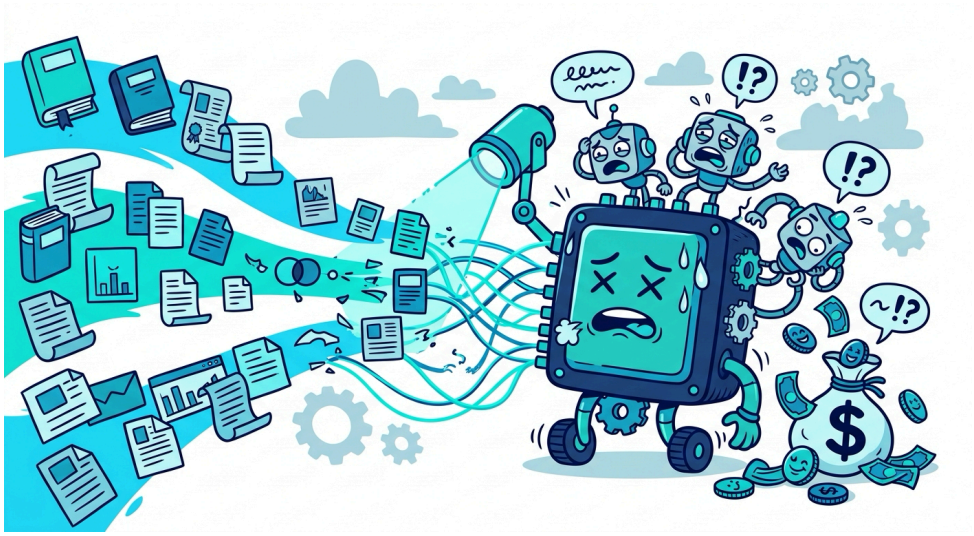

Figure 5: A bloated context window bleeding money and accuracy: the two enemies compression fights

5.1 Cost Is a Function of Tokens, Not API Calls

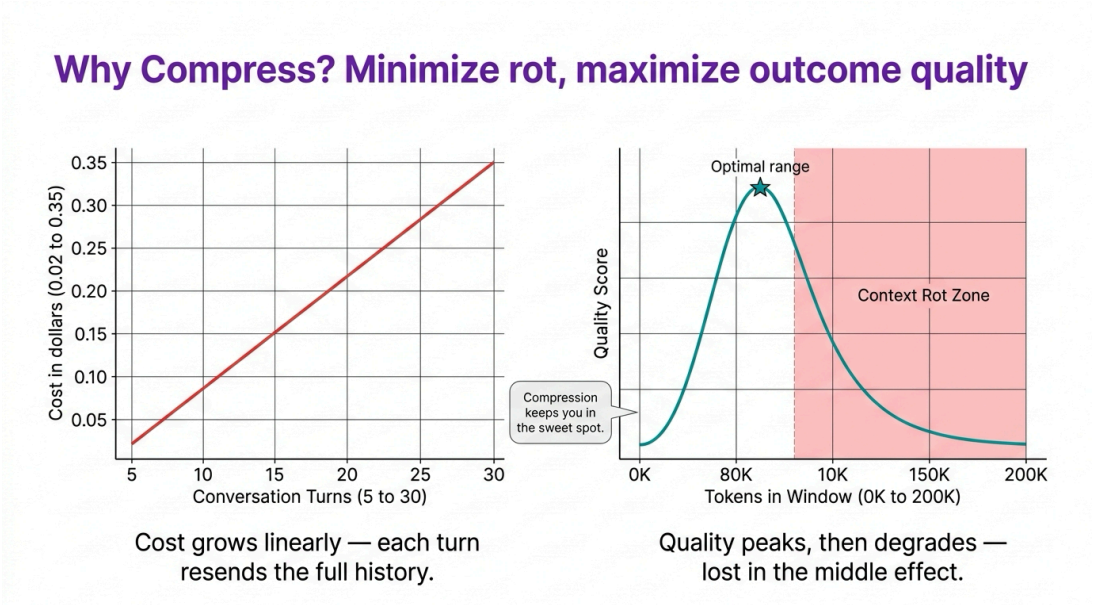

Figure 6: The dual case for compression: cost grows linearly with conversation turns (left), while quality peaks then degrades as tokens accumulate past the optimal range (right). Compression keeps you in the sweet spot.

LLM billing is based on **token consumption**, not the number of API calls. An agent that makes 300 API calls consuming 10,000 total tokens costs less than an agent that makes 20 API

calls consuming 200,000 tokens. This distinction is critical for understanding why compression matters and why sub-agents are not inherently more expensive.

> **Info**
>
> Making 300 separate API calls sounds expensive, but if each call consumes a small, focused context window, the total token cost can be far lower than a single monolithic agent stuffing everything into one 200K-token window.

Consider a single agent working on a large project. Every tool result, every conversation turn, every intermediate output accumulates in the context window. At $3 per million input tokens (Claude Opus pricing), a context window that grows from 50K to 150K tokens over a session costs 3x more per API call — and the cost compounds with every subsequent call.

5.2 Context Rot: The Silent Quality Killer

Cost is only half the story. Even if tokens were free, an overloaded context window would still degrade the model's output quality. This is **context rot** — the phenomenon where the LLM loses track of critical instructions, hallucinates results, and produces contradictory outputs because there is simply too much irrelevant information competing for attention.

> **Danger**
>
> A production agent handling 50-turn customer support conversations will exhibit noticeably confused, contradictory responses after approximately 30 turns if no compression is applied. The context window is not just "full" — it is *rotting*.

The instructor shared a concrete example: when generating a series of 17 images using an AI tool, by the 15th or 16th image the quality degraded significantly because the accumulated tool results from earlier generations were polluting the context. Running `/compact` produced an immediate, step-function improvement in quality.

Table 3: The two reasons compression exists: money and quality

Reason	Mechanism	Consequence
Cost	Token consumption grows linearly with context size; each API call processes the full window	A 100K-token context costs 10x more per call than a 10K-token context
Context rot	Attention mechanism loses focus as irrelevant tokens accumulate	Hallucinated results, contradictory outputs, forgotten instructions

6 Four Types of Context Compression

Claude Code and other modern coding agents employ four distinct compression strategies, each with different trade-offs between compression ratio, information preservation, and computational cost. Before walking through the four, we introduce the metric used to score them.

> **Info**
>
> **A note on running example.** The next several chapters quote concrete numbers from a single recurring scenario — the **BiteBridge** food-delivery incident response. For now, you only need to know that BiteBridge gives us a 1,900-token incident context with 20 identifiable key facts that we apply each technique to. The full scenario — the dinner-rush crisis, the 12 context items, the three sub-agents, and the recovery plan — is unpacked end-to-end in Section 11. When you see "in the BiteBridge example" or "from the BiteBridge simulation" below, that is what we are referring to.

6.1 Information Retention Ratio (IRR): The Quality Metric

The **Information Retention Ratio** is the primary metric for evaluating compression quality. It measures what percentage of key facts survive the compression process.

$$\text{IRR} = \frac{\text{Key facts preserved after compression}}{\text{Total key facts before compression}} \times 100\% \tag{1}$$

> **Example**
>
> A 15-turn operations debug session contains 1,900 tokens with 20 identifiable key facts. After LLM summarization, it becomes 257 tokens with 18 facts preserved. The IRR is $18/20 = 90\%$. After tool result clearing of the same original context, it becomes 1,347 tokens with 20 facts preserved. The IRR is $20/20 = 100\%$.

All compression is lossy. The question is whether what was lost mattered. A compression that drops two trivial facts while preserving all critical ones is superior to a compression that retains everything but achieves minimal space savings.

With IRR in hand, the figure below scores all four strategies on a single axis: 100% means nothing important was lost; 90% means a tenth of the key facts were dropped.

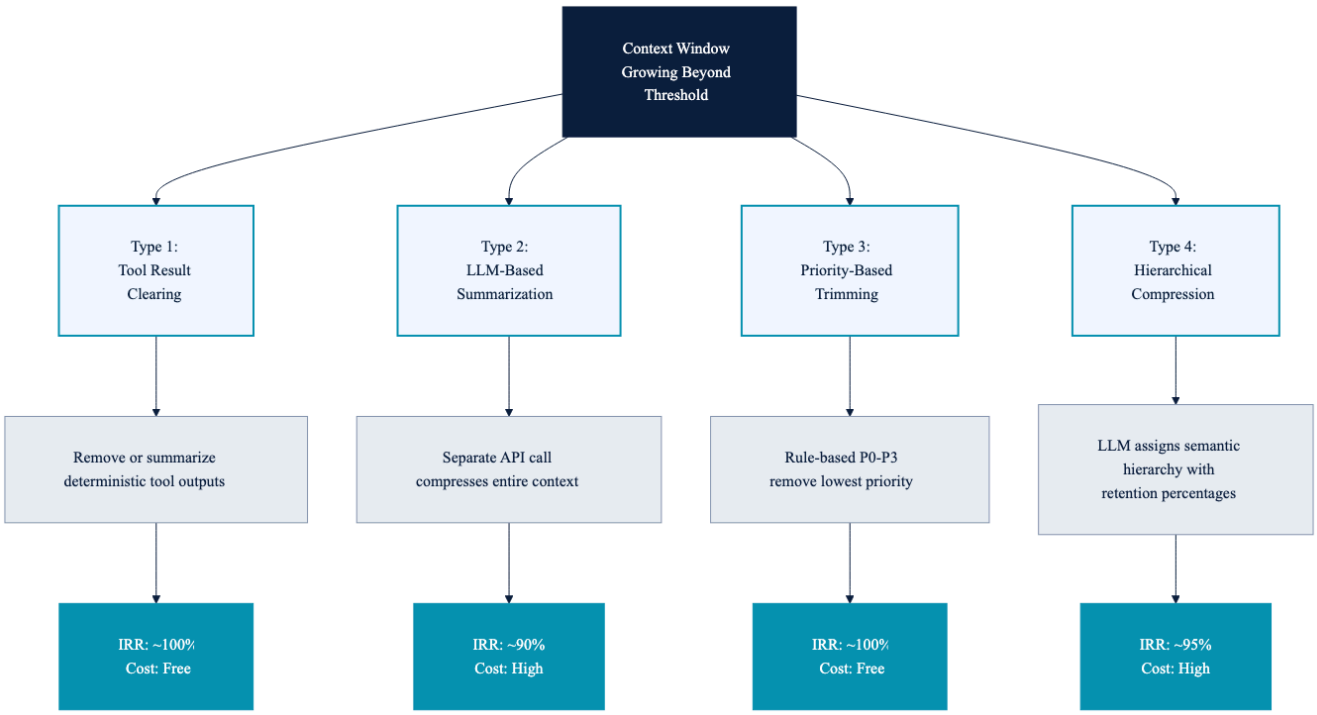

Figure 7: Four compression strategies ordered by aggressiveness, scored by IRR: from gentle tool clearing (100%) to ruthless hierarchical compression

6.2 Type 1: LLM-Based Summarization

The most powerful but most expensive technique. An additional API call is made, passing the current context to the LLM with instructions to summarize it into a much smaller representation.

1. **Detect threshold** — Context window reaches 80% capacity (e.g., 160K out of 200K tokens)

2. **Make summarization API call** — Pass the full context to an LLM with summarization instructions

3. **Receive compressed summary** — The LLM produces a natural language summary (e.g., 10K tokens from 100K)

4. **Replace context** — The compressed summary replaces the original context window contents

The compression is dramatic: in the BiteBridge simulation, LLM summarization compressed 1,900 tokens down to 257 tokens — an 86% reduction. However, the Information Retention Ratio was 90% (18 out of 20 key facts preserved), meaning two facts were lost.

> ⚠️ **Warning**
>
> LLM summarization is lossy. Structure is lost, nuances may not be captured, and quantitative facts can be dropped. The trade-off is massive compression at the cost of some information loss.

The cost of summarization: If your context window has 100K tokens consumed, the summarization API call itself consumes those same 100K tokens as input plus the instruction overhead. You save tokens in future calls, but the summarization call is expensive. This is why you do not compress after every single interaction.

> 🔥 **Tip**
>
> For compression, you do not need your most capable model. A lightweight model that is strong at natural language summarization (not necessarily coding) can handle the compression API call, reducing both latency and cost.

6.3 Type 2: Tool Result Clearing

The safest and most commonly used compression technique. Tool results — the structured JSON responses from function calls — are either entirely removed or compacted into one- or two-sentence natural language summaries.

Why tool results are safe to clear: Tool calls are largely **deterministic**. If a function is called with the same arguments, it produces the same output. If the LLM ever needs a tool result again, it can simply re-invoke the tool. There is no need to store stale results in the context window permanently.

In the BiteBridge simulation, tool result clearing compressed from 1,900 tokens to 1,347 tokens — a more modest reduction. But critically, the IRR was **100%**: all 20 key facts were preserved. The only content removed was structured tool output that could be regenerated.

> ❗ **Memorize**
>
> Tool result clearing should be your **first** compression pass. It is low-risk (results are deterministic and re-fetchable), preserves all reasoning context, and removes the most redundant content. Only escalate to LLM summarization if tool clearing alone is insufficient.

Table 4: LLM summarization vs. tool result clearing: compression ratio vs. safety

Aspect	LLM Summarization	Tool Result Clearing
Compression ratio	Very high (86% in BiteBridge)	Moderate (29% in BiteBridge)
IRR	~90% (some facts lost)	~100% (all facts preserved)
Cost	Expensive (additional LLM API call)	Free (rule-based, no LLM needed)
Risk	Lossy: structure and nuance may be lost	Low: deterministic results can be re-fetched
When to use	After tool clearing is insufficient	First compression pass, always

6.4 Type 3: Priority-Based Trimming

In priority-based trimming, every element of the context window is assigned a priority level using **predefined rules** (not LLM semantics). Elements at the lowest priority are trimmed first — meaning entirely removed, not summarized.

Table 5: Priority levels for trimming: P0 is sacred, P3 is expendable

Priority	Content Type	Action
P0 (highest)	System prompt, core instructions, `CLAUDE.md`	Never removed
P1	Current task context, current user query, current action plan, tool definitions	Preserved; compressed only in extreme cases
P2	Recent conversation history, recent assistant outputs	Compressed if needed
P3 (lowest)	Stale tool results, old conversation history, routine messages	Trimmed first, often entirely removed

In the BiteBridge simulation, priority trimming removed approximately 220 tokens — the routine conversational messages from an operations manager that had low informational value. The IRR remained at 100% because only genuinely low-value content was removed.

> **Info**
>
> Priority-based trimming uses **rules**, not LLM calls. The priority is determined by the *structural position* of the content (system prompt vs. tool result vs. old conversation), not by semantic analysis. This makes it fast and free but less nuanced.

> ⚠️ **Warning**
>
> **Priority trimming is riskier than tool result clearing.** Tool result clearing removes only redundant, deterministic, and re-fetchable data — the reasoning context around the tool call is fully preserved. Priority trimming, by contrast, can delete entire messages (old conversation turns, P3 user inputs) that may contain **implicit dependencies** the current task still relies on — a referenced decision, a constraint mentioned 20 turns ago, an error the user flagged earlier. Always exhaust tool result clearing before resorting to priority trimming.

6.5 Type 4: Hierarchical Compression Using Semantics

The most sophisticated technique. An LLM analyzes the semantic meaning of each section of the context and assigns it to a hierarchy level. Each level receives a different retention percentage:

Table 6: Hierarchical compression: retention percentages decrease down the hierarchy

Hierarchy Level	Retention	Example Content
Critical (top)	100%	System prompt, active instructions, pending user decisions
Important	50%	Tool results from recent calls, recent assistant reasoning
Routine	10%	Older conversation turns, completed task summaries
Low value (bottom)	0%	Stale tool results, routine acknowledgments

The key difference from priority-based trimming: the hierarchy is determined by **semantic analysis** (an LLM decides what is relevant), not by predefined rules. A recent conversation turn might be classified as "low value" if its content has already been captured elsewhere, while an older tool result might be classified as "important" if it contains data the current task depends on.

> 💡 **Idea**
>
> Priority trimming asks: "What *type* of content is this?" Hierarchical compression asks: "How *relevant* is this content right now?" The first uses rules; the second uses LLM judgment. Both reduce context, but hierarchical compression is more nuanced and more expensive.

7 When to Compress: Thresholds and Strategy

Modern coding agents use a **two-threshold system** to trigger compression automatically.

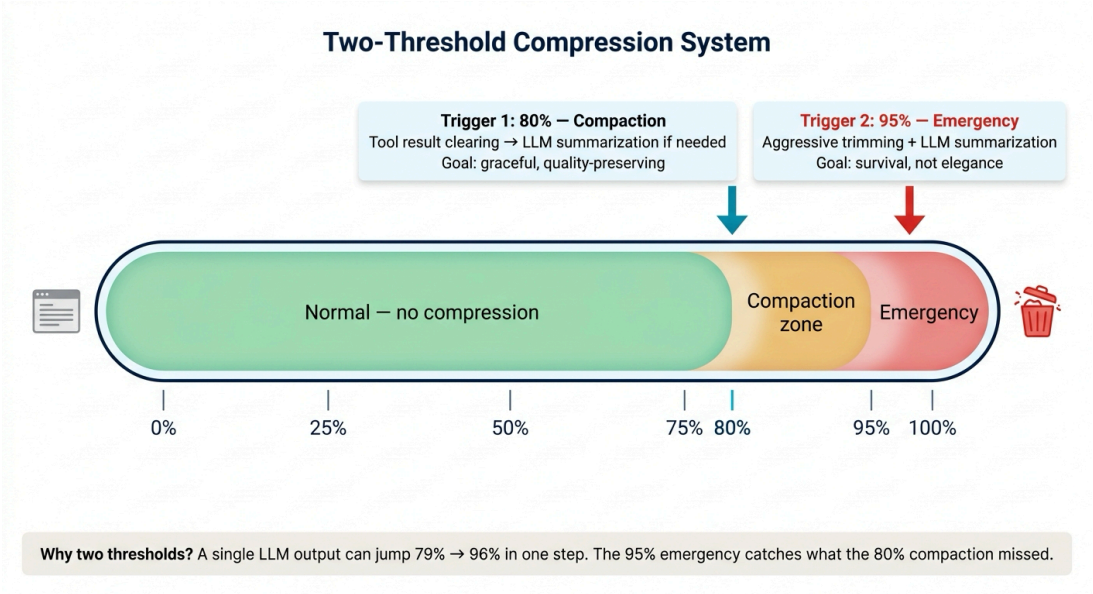

Figure 8: The two-threshold compression system: gentle compaction at 80%, emergency trimming at 95%

7.1 The 80% Compaction Threshold

When the context window reaches **80% capacity** (e.g., 160K out of 200K tokens), auto-compaction triggers. The system first clears tool results, then applies LLM summarization if needed. This is a controlled, quality-preserving operation.

7.2 The 95% Emergency Threshold

If the context window reaches **95% capacity**, emergency trimming fires. This is aggressive: entire tool results are deleted (not summarized), P3 and P2 content is removed, and LLM summarization is applied to whatever remains. The goal is survival, not elegance.

> **? Question**
>
> Why do we need two thresholds instead of just one?

Because a single LLM output can jump the context across both thresholds in one step. If the context sits at 79% of a 200K window and the LLM produces a 35K-token output, the window lands at 96% — the 80% trigger is skipped entirely and emergency trimming fires directly. Without a separate emergency threshold, this scenario would silently overflow and hard-fail the session.

The two-threshold design also supports **multi-pass compaction**. If tool result clearing at 80% only reduces the context to 88%, the system fires pass two (LLM summarization) immediately rather than waiting for the 95% emergency. This graduated approach preserves more information than a single aggressive pass.

Table 7: Two compression thresholds: compaction is careful, emergency is ruthless

Threshold	Trigger	Strategy	Aggressiveness
80% (compaction)	Context reaches 80% of window	Tool result clearing → LLM summarization if needed	Moderate: summarize, don't delete
95% (emergency)	Context reaches 95% of window	Aggressive trimming of all non-P0 content + LLM summarization	High: delete first, summarize remainder

7.3 Practical Compression Workflows

The instructor described two distinct workflows for managing context in practice:

1. `/compact`: When the context is growing from a long operation (e.g., generating many images) but you want to preserve the overall task context. This triggers LLM-based summarization of the existing context.
2. `/clear`: When switching from Task A to Task B with no shared context. This completely resets the context window and provides the best output quality for the new task.

> 🔥 **Tip**
>
> If you observe quality degradation during a long session, run `/compact` and you will see a step-function improvement. If the tasks are completely unrelated, `/clear` provides even better results because the new task starts with a pristine context window.

7.4 `/compact` in Practice: Andrej Karpathy's Auto Research Agent

The most striking real-world example of explicit compression instructions is Andrej Karpathy's **Auto Research** agent [4] — a framework for continuous, autonomous experimentation that ran 24/7 for over two days without a single context-related failure.

The key instruction that made this possible was a single block of natural language inside `CLAUDE.md`:

</> CLAUDE.md instruction for the Auto Research agent

```
Whenever you decide to do a new set of experiments where the
previous experiments are a completely different sandbox, even if
your context window has not reached 80%, run /compact to compress
the context. This ensures baggage from previous experiments is not
carried forward.
```

Because the experiments themselves ran on remote RunPod GPUs, execution logs stayed off-host and never entered the Claude Code context window — a structural reason token consumption stayed low across multi-day runs.

🔥 Tip

For long-running agents, add explicit compression instructions to your `CLAUDE.md` file. Do not rely solely on the 80% auto-compaction threshold. Instruct the agent to compact when switching between logically distinct tasks, even if the context is not yet full.

7.5 What Should Never Be Compressed

Beyond the P0 layer (system prompt, persona, core instructions), there is a subtler category that compression must protect: **state-dependent messages whose meaning depends on a pending interaction**. These are not "important" in the semantic sense — they are *live*, and removing them breaks the application.

The canonical example: **the most recent code edit the assistant has proposed but the user has not yet accepted or rejected.** In a coding-assistant IDE, the diff is shown and the user must click accept/reject before the next step. If auto-compaction collapses that pending diff into "the assistant proposed a refactor," the accept/reject affordance disappears with it — the UI now points at content that no longer exists in context.

Table 8: Compression policy by message category — "live" state is as sacred as the system prompt

Category	Compress?	Reason
System prompt, persona, `CLAUDE.md`	Never	Defines agent behaviour
Pending user decisions (unaccepted edits, awaiting confirmations)	Never	Live UI state; compression breaks the interaction
Active task plan / current step	Never (P1)	Drives next action
Recent tool results	Maybe	Re-fetchable; clear first
Old conversation turns	Yes (low priority)	Information mostly captured downstream
Routine acknowledgements	Yes (first to go)	No informational value

> **Info**
>
> Compression operates on *stable* context. Anything that represents an *in-flight transaction* between the user and the agent — a pending decision, an awaiting confirmation, a half-applied tool call — must be excluded from compression entirely, regardless of its token cost.

8 Sub-Agent Architecture: Context Isolation

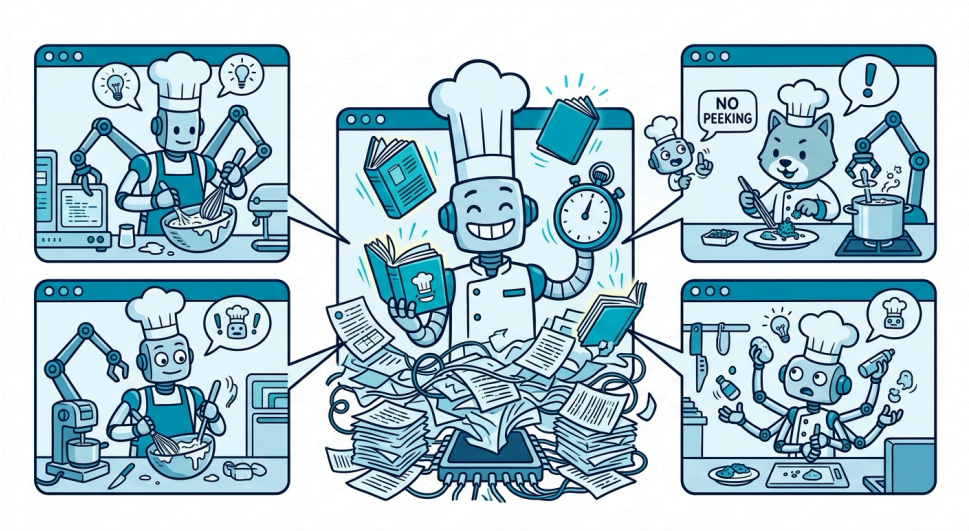

Figure 9: Sub-agents: separate kitchens, separate recipes, one head chef making sure dinner is served on time

The ISOLATE operation in WSCI addresses a fundamentally different problem from compression. Instead of shrinking a single overloaded context window, isolation **distributes** the work

across multiple independent context windows — each belonging to a separate sub-agent with its own API calls.

8.1 Separate API Calls, Separate Context Windows

This is the most important architectural fact about sub-agents: **each sub-agent makes its own, separate API call with its own, separate context window**. Sub-agents do not share memory directly. There is no shared memory bus, no inter-process communication of token state.

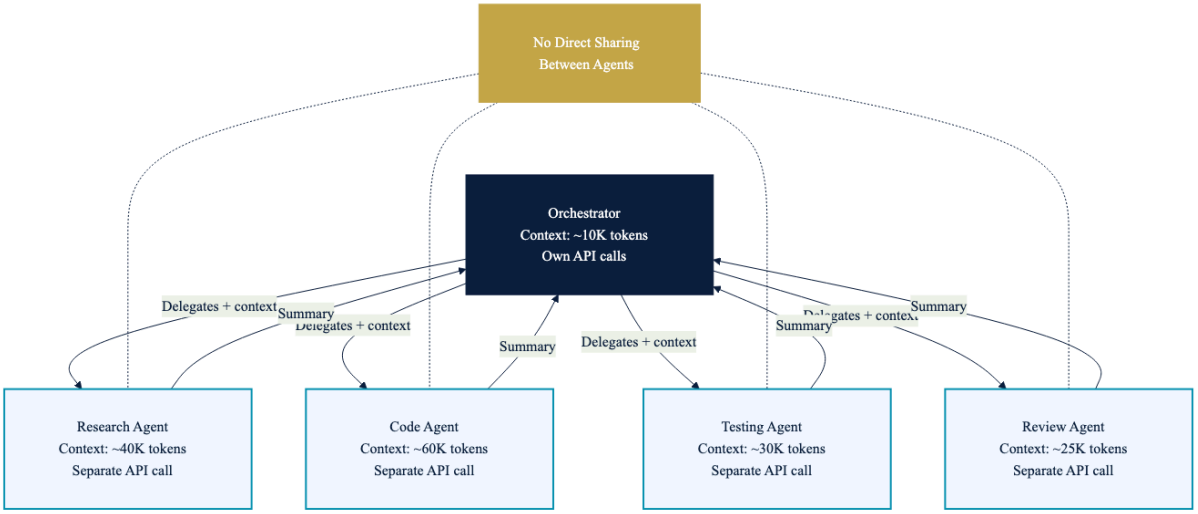

Figure 10: Each sub-agent has its own 200K-token context window and makes independent API calls. No direct memory sharing.

If the orchestrator needs Agent B to know something that Agent A discovered, the orchestrator must explicitly pass that information into Agent B's context. The orchestrator acts as the sole communication bridge between agents.

> **Info**
>
> Sub-agents do not share context windows directly. They may receive the same system prompt (tokenized identically since they use the same LLM and tokenizer), but this is *duplication*, not sharing. All inter-agent communication flows through the orchestrator.

8.2 How Sub-Agents Return Findings to the Orchestrator

If sub-agents have isolated context windows, how do their findings ever reach the orchestrator? The mechanism reuses a basic fact about LLM API calls — that a call returns only its final output, not its internal state — and applies it at the system level:

1. The orchestrator invokes a sub-agent by making an API call (or spawning a sub-agent process).

2. The sub-agent runs to completion in its own context window — which may grow to tens of thousands of tokens during execution.
3. The sub-agent's **final output** (its API response, typically a few hundred tokens) is what the orchestrator receives back.
4. The orchestrator appends that compressed result to its own context window.

Only the final, distilled answer crosses the boundary — intermediate tool results, scratch reasoning, and partial drafts stay inside the sub-agent and are discarded when it terminates.

Table 9: Sub-agent context vs. what reaches the orchestrator: massive asymmetry by design

Quantity	Sub-Agent Side	Orchestrator Side
Total tokens consumed	Up to 30--60K (full investigation)	Only the returned summary (100--300 tokens)
What lives here	Tool results, intermediate reasoning, drafts	Just the conclusion / synthesised finding
Lifetime	Discarded when sub-agent terminates	Persists in orchestrator history

> **Info**
>
> The sub-agent boundary is itself a compression mechanism: a 60K-token investigation collapses into a 200-token summary — roughly 300× compression — which is why ISOLATE and COMPRESS reinforce each other. Every sub-agent acts as a built-in summariser for its own work.

Practical formats in which sub-agents typically return findings:

- **Structured summary:** A short paragraph or bulleted list of conclusions (e.g., "Root cause: misconfigured rate limiter on driver-onboarding endpoint. Evidence: error logs lines 142–189, config audit hash mismatch.")
- **Schema-conformant JSON:** When the orchestrator uses the contract-first pattern, sub-agents return JSON matching a predefined schema (e.g., `{severity: "P1", affected_users: 1240, recommendation: "..."}`).
- **Decision + rationale:** A verdict (approve/reject/escalate) plus a short justification.
- **Pointer to artifact:** For large outputs (a generated file, a database query result), the sub-agent saves the artifact to disk or a shared store and returns only the path or ID.

When several sub-agents run in parallel, the orchestrator must later combine their compressed returns — deduplicating overlapping findings and synthesising a unified result. This step is called **fan-in**, and we cover it in detail in Section 10.

> ⚠ **Warning**
>
> A sub-agent that dumps its full context back to the orchestrator defeats the purpose of isolation. If the Research sub-agent returns a 30K-token raw transcript instead of a 200-token summary, the orchestrator's context grows by 30K — recreating the very rot the sub-agent architecture was meant to prevent. Always instruct sub-agents to produce a *final answer*, not a *full trace*.

8.3 Why Isolation Prevents Context Rot

Consider a project requiring 165K tokens of total work. With a single agent, all 165K tokens accumulate in one context window, causing severe context rot. With sub-agents:

Table 10: 165K tokens distributed across five agents: no single context exceeds 30% utilization

Agent	Role	Tokens Consumed	Context Utilization
Orchestrator	Delegates tasks, synthesizes results	~10K	5% of 200K
Research Agent	Literature review, gap identification	~40K	20% of 200K
Code Agent	Writes implementation code	~60K	30% of 200K
Testing Agent	Executes tests, validates code	~30K	15% of 200K
Review Agent	Reviews code and test results	~25K	12.5% of 200K

The same 165K tokens are consumed overall, but no single agent's context window is overloaded. Each agent operates in a clean, focused context with minimal rot. The LLM does not penalize you for making new context windows — it only cares about total token consumption.

Isolation explains *why* sub-agents help; the next question is *how* each sub-agent learns the rules of the territory it has been spawned into. That is the job of `AGENTS.md`.

9 The `AGENTS.md` Hierarchy

When a project uses sub-agents, the folder layout itself becomes the instruction hierarchy: each directory may carry an `AGENTS.md` file scoped to the work that happens inside it.

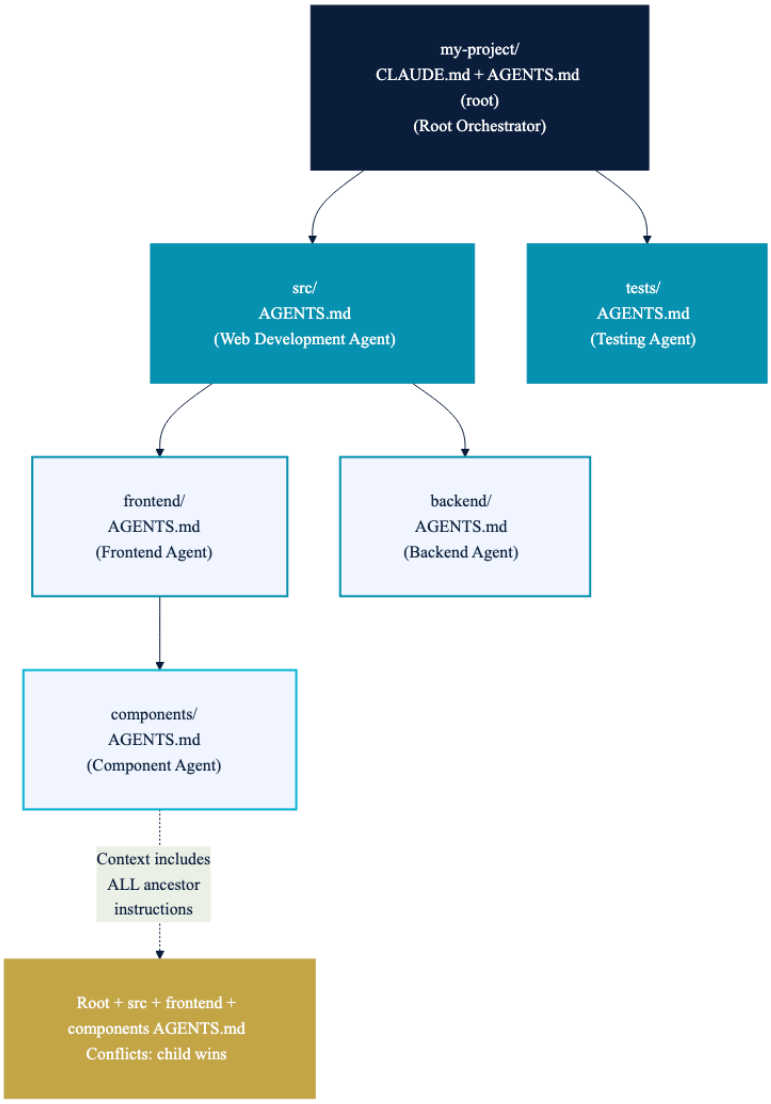

Figure 11: The AGENTS.md hierarchy: each level adds specificity, child instructions override parent on conflict

9.1 Folder Structure Example

Read each comment in the tree below as *"who reads this file, and when"*:

Project structure with CLAUDE.md and nested AGENTS.md files

```
my-project/
├── CLAUDE.md          ← main agent reads this (project-level rules)
├── AGENTS.md          ← sub-agents spawned from root read this
├── src/
│   ├── AGENTS.md      ← sub-agents spawned from src/ read this
│   ├── frontend/
│   │   ├── AGENTS.md  ← sub-agents for frontend code read this
│   │   └── components/
│   │       └── AGENTS.md ← sub-agents for components/ read this
│   └── backend/
│       ├── AGENTS.md  ← sub-agents for backend code read this
│       └── routes/
└── tests/
    └── AGENTS.md      ← sub-agents for tests read this
```

Two rules to remember:

- `CLAUDE.md` is read once by the main chat agent at the start of the session — it is *not* re-read by sub-agents.
- Each `AGENTS.md` is read only when a sub-agent is spawned to work inside that directory (or any of its descendants).

This is why the folder structure doubles as the agent hierarchy: a sub-agent spawned to work inside a deeper folder reads a more narrowly scoped `AGENTS.md`, automatically inheriting the right level of specificity.

9.2 Conflict Resolution: Child Takes Precedence

When instructions conflict between hierarchy levels, the **child-level instruction always wins**. The reasoning is simple: the child agent has more proximity to the task being executed.

Example

If the root `AGENTS.md` says "use a modern frontend style" but the `frontend/AGENTS.md` says "use a legacy monospaced style," the frontend agent follows the legacy instruction. The root instruction is *not* wrong — it is simply *less specific* than the child's.

9.3 Context Accumulation: Concatenation, Not Replacement

A child agent's context includes instructions from **all ancestor levels**, concatenated together. Conflicts are resolved by child precedence; everything else is additive.

Hierarchy resolution walkthrough. Consider a sub-agent working inside `src/frontend/components/`. Here is exactly what it sees, level by level, from broadest to most specific:

1. **Root** `AGENTS.md` (if present)
   → Project-wide rules: *"Use TypeScript strict mode"*
   → Applies to ALL sub-agents everywhere in the project.
2. `src/AGENTS.md` (if present)
   → Source code rules: *"Follow the coding standards in STYLE.md"*
   → Applies to all sub-agents working on source code.
3. `src/frontend/AGENTS.md` (if present)
   → Frontend-specific rules: *"Use React 19, no class components"*
   → Applies to all sub-agents working on frontend code.
4. `src/frontend/components/AGENTS.md` (if present)
   → Component-specific rules: *"Use compound component pattern"*
   → Most specific; highest priority for this directory.

Merged context (in order of specificity):

</> What the components/ sub-agent actually receives

```
Root:        "Use TypeScript strict mode"
src/:        "Follow STYLE.md standards"
frontend/:   "React 19, no class components"
components/: "Compound component pattern"
```

All four instructions land in the sub-agent's context together. If any of them contradict each other, the deeper rule wins — but in practice, each level's rule is scoped to a narrower concern, so conflicts are rare and accumulation is the dominant behaviour. The deeper an agent sits in the hierarchy, the more instructions it accumulates, each relevant to its increasingly specific scope.

9.4 `CLAUDE.md` vs. `AGENTS.md`

The sharpest distinction between `CLAUDE.md` and `AGENTS.md` is *who reads the file* and *when*:

Table 11: `CLAUDE.md` is the workspace constitution; `AGENTS.md` is the per-area onboarding guide

Aspect	`CLAUDE.md`	`AGENTS.md`
Read by	*Main agent* --- your chat session	*Sub-agents* --- spawned by the Agent tool
When loaded	Always loaded at session start	Only loaded when a sub-agent is spawned for work in that directory
Scope	Whole project / workspace	Specific directory and its children
Role	Acts as the *system prompt* / workspace constitution	Acts as the *root-level or folder-level playbook* for that agent
Contains	Project overview, build commands, coding conventions, global rules	Domain-specific architecture, key files and patterns, common pitfalls in this area, testing instructions for this area
Analogy	Company employee handbook --- everyone receives it on day one	Department onboarding doc --- marketing ≠ engineering ≠ sales
Universality	Anthropic-specific (Claude); other platforms use equivalents	Universal across all LLM platforms

Mnemonic: `CLAUDE.md` is *always-on*; `AGENTS.md` is *lazy-loaded*. If a folder is never visited by a sub-agent, its `AGENTS.md` is never read. It costs you no tokens until it earns them.

> **Info**
>
> You can have `AGENTS.md` files without a root-level `AGENTS.md`. If sub-agents only run in a few subdirectories (say, `src/` and `tests/`), put `AGENTS.md` only in those folders. The project root still keeps its `CLAUDE.md` for workspace-wide rules.

10 Multi-Agent Orchestration Patterns

When multiple sub-agents collaborate, the communication pattern between them determines how context flows. Five patterns emerge, each suited to different task structures.

Figure 12: Five ways to wire up a multi-agent kitchen: from total isolation to a tightly sequenced assembly line

10.1 Pattern 1: Contract-First Design

Before sub-agents can collaborate, they need a shared language. The **contract-first pattern** requires the orchestrator to define a common schema that all sub-agents must follow for inputs and outputs.

Example

If a frontend agent and a backend agent are building the same product, the orchestrator defines an `api_specifications.json` that both agents must adhere to. Without this contract, the frontend might call `/api/v1/user` while the backend exposes `/api/users` — incompatible interfaces despite both agents completing their tasks "correctly."

Claude Code implements contract-first implicitly — you do not need to explicitly define schemas for sub-agent communication. However, if you are building custom agent frameworks or using two-way MCP between independently developed agents, explicit contracts are essential.

> ⚡ **Danger**
>
> **The schema must be permissive enough to capture every finding the agent might legitimately produce.** Consider a security sub-agent whose output schema only allows a maximum severity of `warning`. If it discovers five critical SQL injection vulnerabilities, the schema forces them to be downgraded or dropped — the contract has actively destroyed information. Design contracts by enumerating the *worst-case* findings each agent might produce, not the typical case. A contract that cannot represent the truth is worse than no contract at all.

10.2 Pattern 2: Fan-Out / Fan-In

Fan-out is the orchestrator distributing tasks to multiple independent sub-agents simultaneously. **Fan-in** is collecting their results back. The orchestrator is responsible for deduplication and synthesis during fan-in.

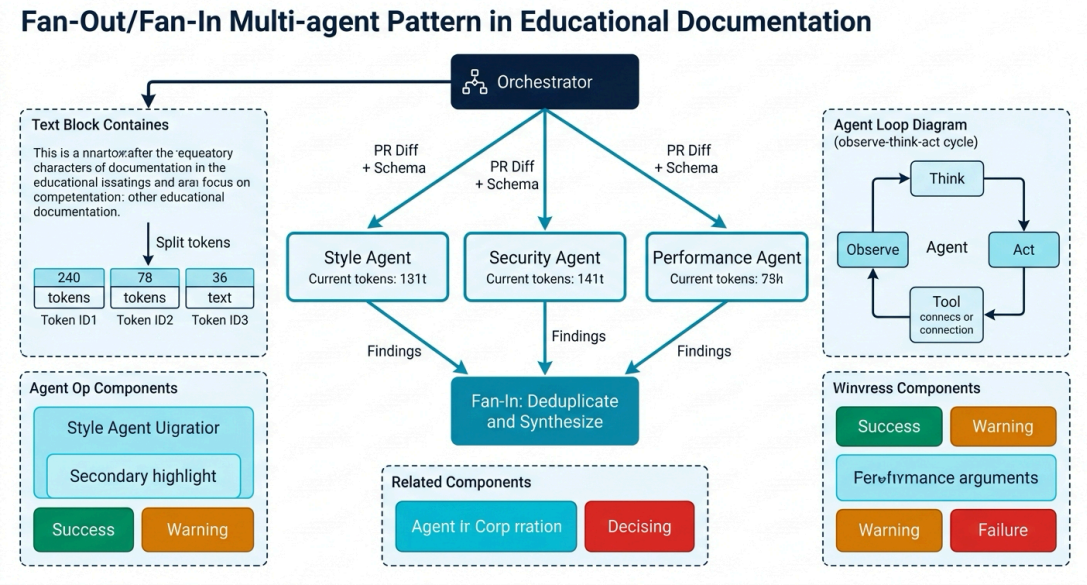

Figure 13: Fan-out: orchestrator distributes. Fan-in: orchestrator collects, deduplicates, and synthesizes.

In a code review scenario, the orchestrator fans out the PR diff to three specialist agents (style, security, performance), then fans in their findings, deduplicates overlapping issues, and produces a unified review verdict.

> ❗ **Memorize**
>
> Fan-out does not necessarily use fewer total tokens. The system prompt is duplicated across all sub-agents. The advantage is not cost savings — it is *specialization*. Each agent applies domain expertise that a single generalist agent would lack.

The orchestrator as a context curator. Fan-out is selective distribution. The orchestrator inspects its pool of available context items and routes only what each sub-agent needs. An Impact Assessor does not need network traces; a Remediation Planner does not need the full config audit. Routing only relevant items keeps each sub-agent's context window lean — the core mechanism behind multi-agent quality gains over a monolithic agent.

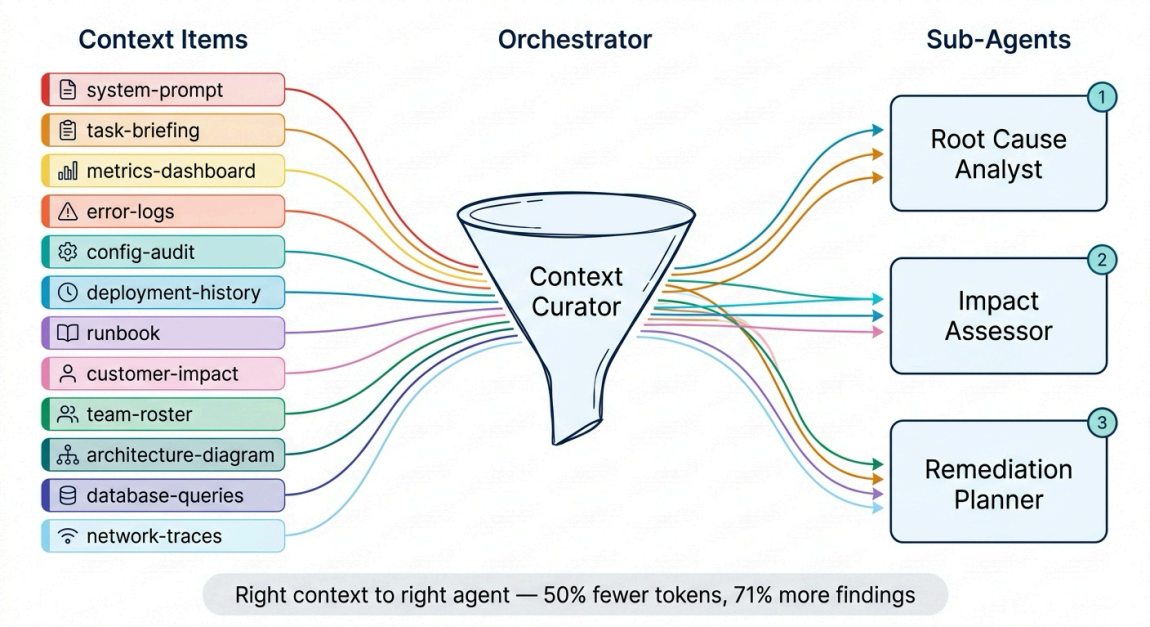

Figure 14: The orchestrator as context curator: twelve items, three specialist sub-agents, selective routing

In the diagram above, notice that **no single sub-agent sees all twelve context items**. The system prompt and task briefing reach every agent (shared base), but specialized items like `network-traces` flow only to the Root Cause Analyst, while `customer-impact` flows only to the Impact Assessor. Each agent operates on a focused slice, so context rot stays low and domain expertise stays sharp.

10.3 Context Sharing: How Agents Communicate

Once sub-agents exist as independent context windows, a second question arises: *what, if anything, do they share?* The contract-first and fan-out/fan-in patterns above govern *coordination* — schema agreement and task distribution. The three patterns below govern *context flow* — how much of one agent's context reaches another. They exist on a spectrum from zero sharing (full isolation) to maximum sharing (sequential pipeline).

10.3.1 Pattern 3: Full Isolation

Each sub-agent receives completely independent context. Beyond the system prompt, agents share no base documents and no common files.

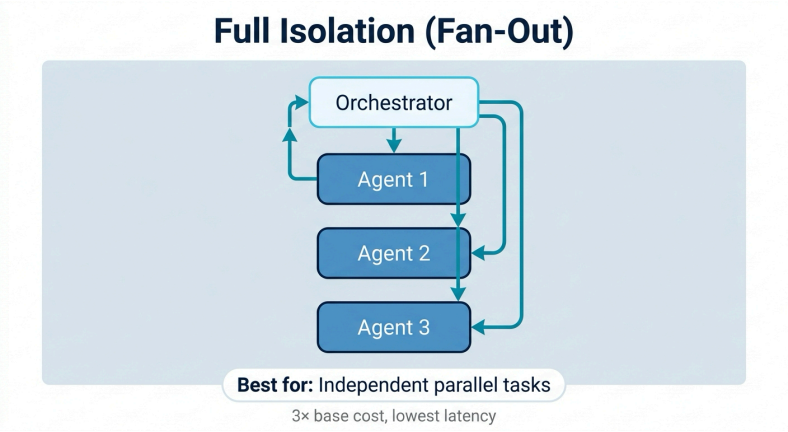

Figure 15: Full Isolation: agents never communicate; 3× base cost, lowest latency

Best for: Tasks where agents analyze entirely different domains with no overlap. Example: parallel city expansion analysis where each agent researches a different city's food delivery market.

10.3.2 Pattern 4: Shared Base Context

Agents share a common foundation (system prompt + shared documents) but receive different specialized context on top. The orchestrator decides which shared documents each agent needs.

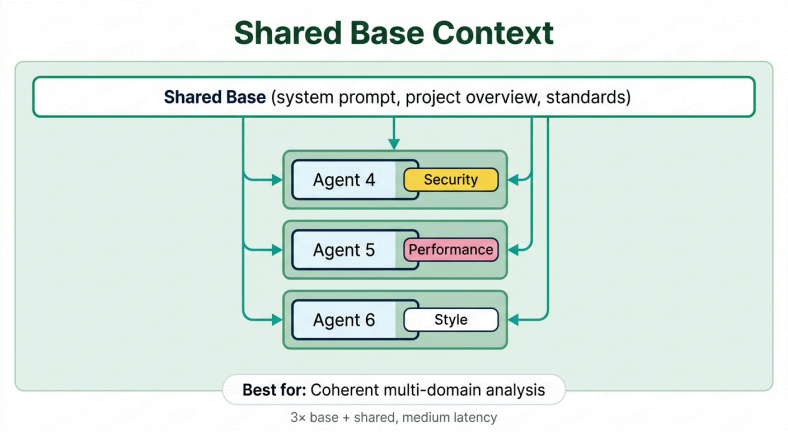

Figure 16: Shared Base Context: common foundation + per-agent specialty; 3× base + shared, medium latency

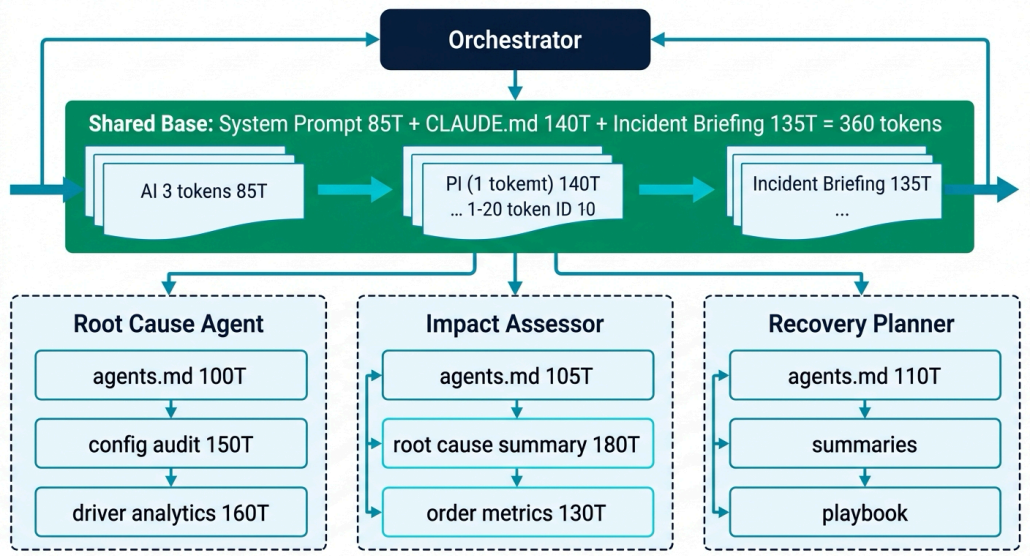

Figure 17: BiteBridge example: 360-token shared base flows to all agents, specialized context layered on top

Best for: Tasks where agents work on the same entity from different angles. Example: multi-domain restaurant onboarding where compliance, menu, and logistics agents all need the restaurant application data but apply different expertise.

In the BiteBridge simulation, the shared base consisted of the system prompt (85 tokens), `CLAUDE.md` (140 tokens), and the incident briefing (135 tokens) — 360 tokens duplicated across all three sub-agents. Each agent then received additional specialized context from the orchestrator.

Total token consumption formula. A common confusion is whether the shared base is paid for once or once per agent. Because each sub-agent makes its own independent API call, the shared base is **re-tokenised in every call**:

$$T_{\text{total}} = \left(T_{\text{shared base}} + T_{\text{specialized}}\right) \times N_{\text{agents}} \tag{2}$$

Example

Three agents have a 1,000-token budget each. The shared base is 300 tokens and each agent receives 600 tokens of specialized context. The total tokens consumed across all agents is *not* $300 + 600 \times 3 = 2\{,\}100$. It is $(300 + 600) \times 3 = 2\{,\}700$ — the 300-token shared base appears in every API call, not once. This is the central trade-off of the shared base pattern: coherence costs duplication.

10.3.3 Pattern 5: Sequential Pipeline

The output of one agent becomes input to the next. Each agent in the pipeline receives a summary of all previous agents' findings plus its own specialized context.

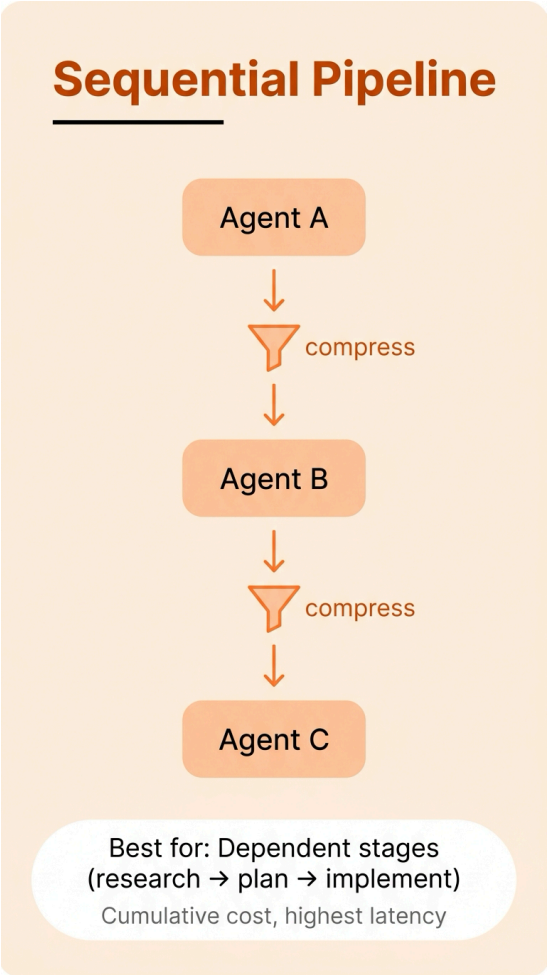

Figure 18: Sequential Pipeline: A ⟶ compress ⟶ B ⟶ compress ⟶ C; cumulative cost, highest latency

Best for: Tasks with natural dependencies. Example: incident response where triage must complete before root cause analysis, which must complete before recovery planning.

Table 12: Sequential pipeline: each agent builds on its predecessors' findings

Agent	Receives	Produces
Triage Agent	System prompt + shared base + incident data	Severity assessment, affected areas
Root Cause Agent	System prompt + shared base + *triage summary*	Root cause identification, evidence
Recovery Agent	System prompt + shared base + *triage summary* + *root cause summary*	Recovery plan, action items

> ⚡ **Danger**
>
> Sequential pipelines enable deeper reasoning but introduce **error propagation risk**. If the triage agent misclassifies the incident severity, the root cause agent builds on a flawed premise, and the recovery agent compounds the error. Guard rails are essential at each pipeline stage.

10.4 Choosing the Right Pattern

If Agents analyze completely different domains with no overlap → **Full isolation** — No shared context needed; maximum independence
If Agents analyze the same entity from different angles → **Shared base context** — Common foundation with specialized expertise
If Agent B cannot start until Agent A finishes → **Sequential pipeline** — Natural dependency chain; each builds on predecessors
If Agents work in parallel but need coordinated outputs → **Fan-out / fan-in with contract-first** — Specialist parallelism with compatible results

11 BiteBridge: Compression and Multi-Agent in Action

Having mapped the five patterns abstractly, we now watch them play out on a single concrete scenario. The rest of this chapter walks through **BiteBridge** end-to-end — the same simulation the instructor demonstrates in the lecture — so a reader who has not seen the video can still follow every token count.

11.1 The Scenario

BiteBridge is a fictional food-delivery company (think DoorDash or Uber Eats). It is **Friday at 7:30 p.m.** — peak dinner rush — and the operations team is suddenly seeing problems:

- **Orders are running late.** Average delivery time has spiked from 28 minutes to 51 minutes in the last hour.
- **Driver analytics show anomalies.** A subset of drivers are stuck with the "available" status but never receive new orders.
- **Customer complaints are climbing.** Support tickets are doubling every fifteen minutes.

A human **operations manager** coordinates the response, but the actual analysis is delegated to a multi-agent AI system. The system has access to a pool of **12 context items** gathered from BiteBridge's internal tools:

Table 13: The 12 context items the multi-agent system can draw from

Context Item	What It Contains
system-prompt	The AI system's persona and operating rules
task-briefing	Plain-English description of the current incident
metrics-dashboard	Real-time order, latency, and throughput numbers
error-logs	Stack traces and warnings from the order-routing service
config-audit	Recent changes to system configuration files
deployment-history	What was deployed in the last 24 hours
runbook	Standard incident-response playbook
customer-impact	Affected customers, complaint volume, sentiment
team-roster	Who is on call, who can be paged
architecture-diagram	How services connect to each other
database-queries	Recent slow queries and lock-wait stats
network-traces	Packet-level traces from the last 30 minutes

The system must produce a **recovery plan** — a short document the human operations manager can act on — in under two minutes.

11.2 Meet the Three Sub-Agents

The orchestrator (the "operations manager AI") delegates the work to three specialised sub-agents in a **sequential pipeline**. Each one has a focused job:

Table 14: Three sub-agents, three focused jobs — the same scenario the instructor uses in the lecture

Sub-Agent	Job	Typical Output
Root Cause Analyst	Read logs, configs, and traces to identify *what broke*	A 1-paragraph diagnosis (e.g., "misconfigured rate limiter on the driver-onboarding endpoint, deployed at 6:48 p.m.")
Impact Assessor	Quantify *who is affected and how badly*	Numbers: late orders, complaining customers, dollar impact, severity grade
Recovery Planner	Draft the *steps to fix the incident*	A numbered runbook the operations manager executes (e.g., roll back deploy, page driver-ops on-call)

Each sub-agent gets only the context items relevant to its job (the fan-out / context-curator pattern from Section 10). The **Root Cause Analyst** needs `error-logs`,

`config-audit`, and `network-traces`; the **Impact Assessor** needs `metrics-dashboard` and `customer-impact`; the **Recovery Planner** needs `runbook` and the previous two agents' outputs.

11.3 Compression Comparison

Before walking through the multi-agent flow, we first ask a simpler question: *if a single agent had to handle this incident, how would compression alone perform?* The full incident context runs 1,900 tokens and encodes 20 key facts (concrete claims like "rate-limiter QPS = 50", "deploy at 6:48 p.m.", "12 drivers stuck") used to score retention. Four compression techniques are applied to it:

Table 15: Compression on the BiteBridge incident: output size, ratio, and IRR

Technique	Output Tokens	Compression	IRR
Original (no compression)	1,900	0%	100%
LLM Summarization	257	86%	90%
Tool Result Clearing	1,347	29%	100%
Priority Trimming (P3 only)	~1,680	~12%	100%
Hierarchical Compression	~800	~58%	~95%

Recommended production sequence: (1) clear tool results first — safe, free, 100% IRR; (2) apply LLM summarization if still above threshold — lossy but high compression; (3) fall back to priority trimming before the 95% emergency threshold kicks in.

11.4 Multi-Agent Incident Response

Now we run the full multi-agent pipeline on the same incident. Each sub-agent runs in turn, builds on the previous one's findings, and returns a compressed summary to the orchestrator:

1. **Root Cause Analyst** (runs first).

   Receives: system prompt (85 tokens) + `CLAUDE.md` (140 tokens) + incident briefing (135 tokens) + its own `AGENTS.md` (100 tokens) + orchestrator-delegated context items: `config-audit` (150 tokens) + driver analytics from `metrics-dashboard` (160 tokens) = **770 tokens total**.

   Returns: a 180-token diagnosis summary to the orchestrator.

2. **Impact Assessor** (runs second).

   Receives: the shared base (360 tokens) + its own `AGENTS.md` (105 tokens) + the Root Cause Analyst's summary (180 tokens) + `metrics-dashboard` order numbers (130 tokens) +

baseline comparison data (90 tokens) = **870 tokens total**.
Returns: a 180-token impact assessment to the orchestrator.

3. **Recovery Planner** (runs last).
   Receives: the shared base (360 tokens) + its own `AGENTS.md` (110 tokens) + the previous two summaries + recovery metrics + customer playbook = **960 tokens total**.
   Returns: the final recovery plan to the orchestrator.

Each hand-off compresses the previous agent's full investigation context (which may have grown to many thousands of tokens internally) down to a 180-token summary. The orchestrator's own context grows by only that much after each step. After all three agents finish, the orchestrator holds the full recovery plan — having consumed only 1,000 tokens in its own window while the sub-agents collectively processed roughly 3,600 tokens (computed below).

Example

A typical recovery-plan output the orchestrator would emit: "Rollback `driver-onboarding-svc` to commit `a1b2c3d` (deployed 18:48). Page on-call SRE Maya Patel. Re-queue 312 stuck orders via `requeue-tool`. Estimated time-to-recovery: 8 min. Customer comms via support template T-117."

11.5 Token Distribution Across Agents

Tallying the BiteBridge numbers across all three sub-agents reveals the central trade-off of multi-agent isolation:

Table 16: Per-agent token totals in the BiteBridge incident response (numbers in tokens)

Agent	Shared Base	Specialized Context	Total
Root Cause Analyst	360	410	770
Impact Assessor	360	510	870
Recovery Planner	360	~600	~960
Orchestrator (final)	—	~1,000	~1,000

The shared base (system prompt + `CLAUDE.md` + incident briefing = 360 tokens) appears in **every** sub-agent's context. This is *not* shared memory — it is the same content tokenized independently in each API call. The total context consumed across all sub-agents is therefore:

$$T_{\text{total}} = \sum_{i=1}^{N} \left(T_{\text{shared base}} + T_{\text{specialized}_i} \right) \approx 3,600 \text{ tokens} \tag{3}$$

Compare this to a hypothetical single-agent solution that would have all 3,600 tokens piled into one context window — with the orchestrator additionally needing to accumulate every intermediate finding, easily pushing past 5K tokens for the same task.

The multi-agent system uses *slightly more* total tokens than a single-agent solution (because the shared base is duplicated), but each individual context window stays under 1K tokens — well below any context-rot threshold. You pay a small token premium and buy a large quality margin. This is the trade-off ISOLATE makes: a few extra tokens for dramatically better focus per call.

12 Glossary

Term	Definition
Auto-compaction	Automatic context compression triggered when the context window reaches a predefined threshold (typically 80%)
Central Brain pattern	Multi-agent orchestration architecture where independent agent projects are connected through a central orchestrator via two-way MCP
Context rot	Progressive degradation of LLM output quality as irrelevant tokens accumulate in the context window
Contract-first pattern	Multi-agent design where the orchestrator defines a shared schema that all sub-agents must follow
Emergency trimming	Aggressive context reduction triggered at 95% capacity; removes entire sections rather than summarizing
Fan-in	The orchestrator collecting and synthesizing results from multiple sub-agents
Fan-out	The orchestrator distributing tasks to multiple independent sub-agents simultaneously
Full isolation	Multi-agent pattern where sub-agents share no context beyond the system prompt
Hierarchical semantic compression	LLM-based compression where semantic analysis assigns retention percentages to different context sections
Information Retention Ratio (IRR)	Percentage of key facts preserved after compression: preserved facts / total facts × 100%
Priority-based trimming	Rule-based context reduction where content is assigned priority levels (P0--P3) and lowest priority is removed first
Sequential pipeline	Multi-agent pattern where each agent's output feeds into the next agent's input
Shared base context	Multi-agent pattern where agents share common documents plus agent-specific specialized context
Sub-agent	An independent agent spawned by an orchestrator with its own API calls and separate context window
Tool result clearing	Removing or summarizing deterministic tool outputs from the context window

Two-way MCP	Bidirectional MCP connection where both orchestrator and sub-agents can initiate calls --- enabling the Central Brain to delegate tasks and receive status updates over the same protocol

13 Notation

Symbol	Meaning
IRR	Information Retention Ratio
P0, P1, P2, P3	Priority levels for trimming (P0 = highest, P3 = lowest)
T	Total tokens in context window
T_{max}	Maximum context window size (e.g., 200K for Claude)
T_{compact}	Compaction threshold (typically 0.8 × T_max)
$T_{\text{emergency}}$	Emergency trimming threshold (typically 0.95 × T_max)

14 Open Questions

1. **Optimal compression model selection:** Which model does Claude Code use for the auto-compaction call — the active session model (e.g., Opus) or a cheaper one (Haiku/Sonnet)? *Research: Anthropic documentation on compaction internals.*
2. **Keyword-based compression:** The instructor mentioned extracting required portions via keyword matching before passing to an LLM for compression — reducing the summarization input from 120K to 30K tokens. *Explore: hybrid keyword + LLM compression pipelines.*
3. **Thinking tokens and context:** Are chain-of-thought / reasoning tokens included in the context window for subsequent calls? The instructor suggested yes, but this may vary by implementation. *Verify: Anthropic documentation on thinking token handling.*
4. **Error propagation in sequential pipelines:** What guard rails are effective for preventing cascading errors when early agents in a pipeline produce flawed outputs? *Research: validation patterns for multi-agent sequential workflows.*
5. **Cross-framework universality:** While `AGENTS.md` is described as universal across all LLM platforms, how do non-coding frameworks (LangGraph, CrewAI, Google ADK) implement equivalent compression and isolation? *Explore: compression APIs in LangGraph and CrewAI.*
6. **Central Brain scalability:** How does the orchestrator handle 10+ agents? Does it need its own tool selection / RAG mechanism for choosing which agent to delegate to? *Research: hierarchical orchestration with sub-orchestrators.*
7. **Two-way MCP state synchronization:** When a sub-agent updates shared state mid-task, how does the Central Brain learn about it — push notifications, polling, or shared storage? *Read: MCP specification on server-initiated notifications.*
8. **MCP authentication for agent networks:** How to implement secure inter-agent communication in the Central Brain pattern? Each agent may need different access levels. *Critical for enterprise deployments.*

References

[1] LangChain, "The WSCI Framework: Write, Select, Compress, Isolate." [Online]. Available: https://blog.langchain.dev/context-engineering-for-agents/

[2] Anthropic, "Model Context Protocol." [Online]. Available: https://modelcontextprotocol.io/

[3] OpenClaw, "OpenClaw: Personal AI Assistant Orchestrator." [Online]. Available: https://github.com/openclaw

[4] A. Karpathy, "Auto Research: Continuous AI-Driven Experimentation." [Online]. Available: https://github.com/andrejkarpathy/auto-research

[5] Anthropic, "Claude Code: Agentic Coding with Claude." [Online]. Available: https://docs.anthropic.com/en/docs/claude-code

[6] Anthropic, "Context Management in Claude Code." [Online]. Available: https://docs.anthropic.com/en/docs/claude-code/memory

[7] S. Panat and R. Dandekar, "LLM Context Engineering Bootcamp." [Online]. Available: https://www.youtube.com/playlist?list=PLPTV0NXA_ZSj-E8A1DBvbWIYGxC46u-Hd

Follow me for More AI Content

If you found these notes useful, connect with me on LinkedIn for more deep dives into Machine Learning, Artificial Intelligence, and Computer Vision.

Isham Rashik on LinkedIn

Scan the QR code or click the link above