Context Engineering & Reliability by Michaël Bettan

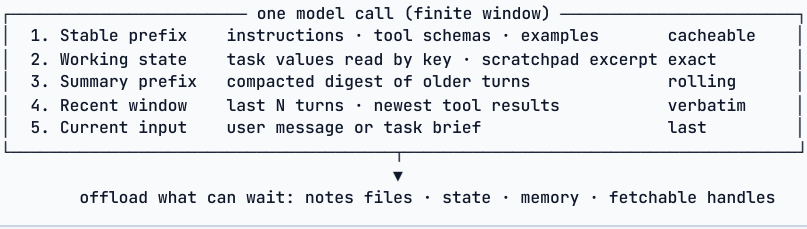
Definition

Context engineering is the per-call discipline of choosing which tokens enter a model's finite context window (instructions, tool definitions, working state, retrieved material, history and tool results), in what form and in what order, and what is moved outside the window so a long-running agent stays coherent after hundreds of turns.

What context engineering solves for

A model is stateless: at each step it sees only what the harness placed in the window for that call. Agents running in a loop keep producing tokens (tool payloads, reasoning, drafts) and most of them stop mattering a few turns later. Left alone, the window fills, recall of the details that still matter drops, and the agent either hits the hard limit or slowly drifts away from its instructions. Context engineering keeps the window small, current and high-signal for the whole run.

> **MENTAL MODEL**
> treat the window like a kitchen counter during a long dinner service. Only the ingredients for the dish being plated stay on the counter; finished pans go to the sink (clearing), prep notes go on the whiteboard (scratchpad), the pantry is opened only when a recipe calls for something (just-in-time retrieval), and a second cook works a separate station (subagent).



- **Scope of this guide:** budget and placement, long-session techniques (sliding windows, summary prefixes, compaction), tool-result clearing, scratchpads and structured note-taking, just-in-time retrieval, sessions and memory, subagent context isolation, strategy selection, caching and security.
- **Vendors covered:** Claude (Messages API context management, memory tool, Agent SDK, Claude Code) and Google (ADK compaction and state, Agent Platform Sessions and Memory Bank, Gemini API caching). Mechanisms differ; the design decisions are shared.

Prompt vs. Context vs. Memory

Layer/approach	Type	Role & behavior
Prompt engineering	Authoring	Wording, structure and examples of one instruction; written once, reused on every call
Context engineering	Per-turn curation	Selects, orders, compresses, offloads and isolates everything in the window on every call
State	Exact working data	Key/value values the harness reads and writes deterministically (IDs, flags, step outputs)
Scratchpad	Agent-written notes	Files the agent writes and re-reads during a task (plan, progress, open questions); outlives a context reset, usually not the task
Memory	Long-term recall	Facts extracted across sessions and fetched by search or a memory tool; recall is probabilistic and can be stale
Retrieval (RAG)	Shared knowledge	Documents valid for every user (policies, manuals), fetched up front or on demand

Context levers

- **Trim** drops the oldest turns once the window passes a size (a sliding window over history).
- **Summarize** replaces a span of older turns with a digest that sits in front of the recent turns (summary prefix, compaction).
- **Clear** removes stale tool results or thinking blocks while leaving the rest of the conversation intact (context editing).
- **Offload** writes information to a file, state or memory and keeps only a handle (path, key, query) in the window.
- **Fetch late** keeps references instead of payloads and loads data with a tool when a step needs it (just-in-time retrieval).
- **Isolate** gives a noisy subtask its own window (subagent or agent-as-tool) and brings back a short digest.
- **Mapping to the common framing:** offload corresponds to *write*, fetch late to *select*, trim, summarize and clear to *compress*, and isolate to *isolate* (the four strategies in the Agentic design patterns guide).

> **KEY RULE**
> every token draws on the same attention budget. A bigger window raises the ceiling, but irrelevant tokens still lower recall of the relevant ones, so the goal is the smallest set of high-signal tokens for the current step.

Per-turn assembly workflow

1. **Load:** read session events, the working state for this task and, if the task uses one, the current scratchpad.
2. **Reduce:** apply the history policy (sliding window, summary prefix or compaction) and clear tool results that can be re-fetched.

3. **Fetch:** pull in only the documents or memories the next step needs, or expose search tools and let the agent decide.
4. **Place:** stable prefix first, then working state and the summary prefix, then recent turns, then the current input.
5. **Call:** count tokens when close to the limit and reserve room for the output.
6. **Offload:** after the call, write new decisions to state or the scratchpad and replace bulky results with handles.

When not to engineer context

- **Short, single-turn tasks:** one well-written prompt with the needed document is enough; compaction, notes and memory add cost with no benefit.
- **Everything fits and stays relevant:** a 20-turn chat with small tool results needs no trimming; measure before adding machinery.
- **Shared reference knowledge:** put it in retrieval or the stable prefix, not in per-user memory.

Context budget and placement

Budget arithmetic

- **The window holds input and output:** system prompt, tool definitions, history, tool results and the reply all count; reserve output headroom (`max_tokens` on Claude, `max_output_tokens` on Gemini) before filling the rest.
- **Window sizes differ by model:** current Claude models default to a 1M-token window with up to 128k output tokens, while legacy models keep 200k; check the model page rather than assuming.
- **Count, don't guess:** Claude exposes a token counting endpoint (`POST /v1/messages/count_tokens` , returning `input_tokens`) and Gemini exposes `count_tokens` ; character heuristics (about four characters per token in English) are only rough planning numbers.
- **Recount when you switch models:** tokenizers change between model generations (Anthropic documents roughly 30% more tokens for the same text with the tokenizer introduced in Opus 4.7), so a prompt that fit before can overflow or cost more after an upgrade.
- **Let the agent see its budget:** recent Claude models receive a context-awareness note after tool calls with tokens used and remaining, which helps them wrap up or write notes before the limit.
- **Tool definitions are a fixed tax:** every tool schema is sent on every call. A bloated tool set also creates ambiguous choices; if an engineer cannot say which tool fits a situation, the model cannot either. Keep a minimal set per agent.
- **Context rot:** recall of a specific detail degrades gradually as total tokens grow, well before the hard limit. Treat the usable window as smaller than the advertised one.

Placement

- **Position effect:** models use information at the start and end of a long input more reliably than material buried in the middle (Liu et al.). Keep the instruction that governs the step near an edge.
- **Long documents first, question last:** for large inputs, place documents above the instructions and the question at the end; Anthropic's long-context guidance reports better answers with this order.
- **Label what you place:** wrap documents and notes in named sections or XML-style tags (`<notes>` , `<document source="..." date="...">`) so the model can refer to them precisely.
- **Prompt altitude:** write instructions specific enough to steer behavior but general enough to act as heuristics. Hard-coded if-else rules are brittle; vague guidance assumes context the model does not have.
- **Examples over rule lists:** a few diverse, canonical examples steer more reliably per token than a long list of edge-case rules.

> [!NOTE] *Directional figures:* the "about four characters per token" heuristic and vendor-reported quality gains from document placement are rules of thumb. Measure on your own prompts with the token counting APIs and an evaluation set.

Long sessions: sliding windows, summary prefixes and compaction

Symptoms of an overloaded session

- **Generic answers:** the agent talks about "the usual approach" instead of the specific file, ticket or decision it found earlier.
- **Instruction drift:** a rule stated early stops being applied after many large tool calls.
- **Repeated work:** the agent re-runs searches whose results it already has, because those results are buried or were dropped without a trace.

History policies

Policy	How it works	Keeps	Loses	Fits
Sliding window (trimming)	Send only the last N turns or last K tokens; drop older turns	Recent detail, verbatim	Everything older, abruptly	Chat where old turns rarely matter
Summary prefix	A rolling digest of older turns is placed before the recent window and refreshed as turns age out	Gist of older turns plus recent detail	Exact wording of summarized turns	Assistants with long back-and-forth
Compaction (reset with summary)	Near a threshold, summarize the whole conversation and continue in a fresh window seeded with the summary	Decisions, open issues, key artifacts	Redundant tool output and chatter	Long agentic runs with tool use
Overlapping compaction	Each new summary covers a span that overlaps the previous one	Continuity across summary boundaries	Some extra tokens	Event-based compaction in ADK

- **Sliding windows are cheap but blunt:** they need no extra model call, yet a constraint stated in turn 3 disappears at turn 3 + N. Anything that must survive belongs in the system prompt, state or a notes file, never only in early turns.
- **Summary prefixes trade fidelity for continuity:** the digest is generated by a model, so it paraphrases. Keep values that must stay exact (amounts, IDs, deadlines) in state or a scratchpad that the summarizer never rewrites.
- **Compaction is the usual first lever for long agent runs:** Anthropic describes Claude Code compaction as summarizing the history while keeping architectural decisions, unresolved bugs and implementation details, discarding redundant tool output, and continuing with the summary plus the five most recently accessed files.

Tuning a compaction prompt

1. **Collect real traces:** pick long, messy runs where a later step depended on an early detail.
2. **Maximize recall first:** write the summary instructions so nothing that a later step used is dropped.
3. **Then raise precision:** remove categories of content the agent never needed again (raw tool payloads, superseded drafts).
4. **Re-test on new traces:** overly aggressive summaries lose details whose importance appears only later.

Vendor mechanisms

Mechanism	Identifier and key parameters	Behavior
Claude on-demand compaction	Top-level `compaction: {"type": "summarize", "instructions": ...}` (beta `compact-2026-09-04`)	Your app decides when; Claude returns a signed `compaction` block that you send first in later `messages`, in place of the summarized turns
Claude threshold compaction	Edit `compact_20260112` (beta `compact-2026-01-12`); `trigger` defaults to 150,000 input tokens; optional `instructions`; `pause_after_compaction`	Server summarizes when input crosses the trigger; pausing returns `stop_reason: "compaction"` so you can inspect the summary or re-insert notes
Claude Code	Auto-compact (`autoCompactWindow` setting), `/compact [focus]`, `/context`, `/clear`	Project-root `CLAUDE.md` is re-read from disk after compaction, so durable rules belong there
Agent SDK	`compact_boundary` system message with `compact_metadata` (`pre_tokens`, `trigger`)	Lets the harness log compactions and re-inject state afterwards
ADK event compaction	`EventsCompactionConfig(compaction_interval, overlap_size, summarizer)` on `App`; token mode uses `token_threshold` with `event_retention_size`	Every `compaction_interval` completed events, older events are summarized; `overlap_size` re-includes earlier compacted events so summaries overlap; recent events stay verbatim

- **Choose the summarizer model deliberately:** ADK accepts a custom `LlmEventSummarizer` with its own model, so a smaller, cheaper model can write summaries while the main agent runs on a larger one.
- **Steer what survives:** both Claude compaction modes accept `instructions`, `/compact` accepts a focus string and ADK accepts a custom summarizer prompt. Name the categories to keep verbatim (open decisions, file paths, user commitments).

> COMMON TRAP
> relying on early turns to carry a rule once a sliding window or compaction is enabled. Restate durable rules in the system prompt or a re-read file such as `CLAUDE.md`, so trimming can never remove them.

Tool-result clearing and context editing

- **Lightest-touch compaction:** once a tool result has been read and acted on, the agent rarely needs the raw payload again. Clearing it frees space without asking a model to rewrite anything, which is why Anthropic describes tool-result clearing as one of the safest forms of compaction.
- **Calls stay visible:** by default Claude clears only the tool results and keeps the original tool calls, so the agent still knows what it asked for and can call the tool again if it needs the data.
- **Token-efficient tools reduce the need to clear:** tools that paginate, filter on the server and return handles to large objects put fewer tokens in the window to begin with.

Claude context editing strategies

Strategy	Key parameters	Behavior
Tool-result clearing (`clear_tool_uses_20250919`, beta `context-management-2025-06-27`)	`trigger` (when clearing starts), `keep` (recent tool uses kept, default 3), `clear_at_least`, `exclude_tools`, `clear_tool_inputs`	Replaces older tool results with placeholders once the prompt crosses the trigger
Thinking-block clearing (`clear_thinking_20251015`)	`keep: {"type": "thinking_turns", "value": N}` or `"all"`	Drops thinking blocks from older assistant turns; listed first in `edits`

```
"context_management": {"edits": [{
  "type": "clear_tool_uses_20250919",
  "trigger": {"type": "input_tokens", "value": 60000},
  "keep": {"type": "tool_uses", "value": 4},
  "exclude_tools": ["memory"]
}]}
```

- **Protect tools whose results stay relevant:** list them in `exclude_tools` (a memory tool, a configuration lookup that the whole task depends on).
- **Pair clearing with notes:** when the memory tool is enabled and the context approaches the clearing threshold, Claude receives an automatic warning so it can save important information to memory files first, then look it up later instead of keeping it in the window.

- **Mind the cache:** clearing tool results invalidates the cached prefix from the cleared point. Use `clear_at_least` so each clearing event frees enough tokens to justify the cache write.
- **ADK equivalent:** there is no built-in clearing strategy; prune or rewrite history in a `before_model_callback`, or store bulky outputs as artifacts and keep a reference in the event.

Scenario: shipment tracing at Larchmont Freight

A tracing agent calls a carrier API about 40 times per investigation; each response is a large JSON document.

- **Before:** after 25 calls the window is mostly raw carrier responses and the agent stops applying the customer's "ground only" constraint.
- **After:** tool-result clearing triggers at 60k input tokens and keeps the last four results; the `shipment_policy` lookup is excluded from clearing; the agent writes each confirmed scan event to a `trace.md` notes file as it goes.
- **Outcome:** the window stays under the trigger for the rest of the run, the constraint stays visible and earlier findings remain one file read away.

> **COMMON TRAP**
> clearing with no breadcrumb. If an agent sees neither the result nor a placeholder or note, it assumes the lookup never happened and repeats it.

Scratchpads and structured note-taking

- **Definition:** the agent periodically writes notes to storage outside the window (a `NOTES.md`, a to-do list, a progress file) and reads them back when it needs them, typically after a compaction, a context reset or a new session.
- **Why it works:** notes give persistence with little overhead. A short progress file carries goals, completed steps and dependencies across dozens of tool calls that would otherwise be trimmed or summarized away.
- **Agent-written, agent-read:** unlike state, which the harness owns, the scratchpad is curated by the agent itself, which decides what is worth remembering.

```
# trace.md (agent scratchpad)
Goal: locate pallet LF-88213; customer requires ground transport
only
Done: origin scan Reno 11-02 06:10; hub scan Denver 11-03 21:44
Open: no scan since Denver; carrier ticket CT-5512 pending
Next: query Denver hub scan log; re-check ticket at 14:00
```

Ephemeral vs. Durable notes

Kind	Lifetime	Examples	Clean-up
Ephemeral scratchpad	One task or run	Working plan, intermediate tallies, hypotheses being tested	Delete or archive when the task closes
Task progress file	Until the task completes, across resets and sessions	`progress.md`, to-do list, test status	Removed at completion
Durable memory	Across tasks and sessions	User preferences, project conventions, lessons learned	Consolidated and expired by policy

Vendor mechanisms

- **Claude memory tool** (`memory_20250818`) is client-side: Claude issues `view`, `create`, `str_replace`, `insert`, `delete` and `rename` commands against a `/memories` directory that your application stores. With the tool enabled, Claude is instructed to check the directory before starting and to record progress as it works.
- **Claude Code** keeps durable project guidance in the `CLAUDE.md` hierarchy and auto memory (`MEMORY.md`); a plain notes file in the repository works as a task scratchpad.
- **ADK** offers `temp:` state for values that live only within one invocation, session state for task progress, and artifacts (`Artifact Service`) for larger, versioned notes files.

Note-taking workflow

1. **Seed:** at task start, create the notes file with the goal and constraints.
2. **Append at milestones:** after each confirmed finding or decision, write one line; avoid logging every tool call.
3. **Re-read at boundaries:** after compaction, clearing or a new session, read the file before acting.
4. **Prune:** collapse completed sections so the file itself stays small.
5. **Close out:** delete the ephemeral parts and promote only durable lessons to long-term memory.

- **Anti-pattern (notes as a transcript):** copying raw tool output into the scratchpad recreates the bloat the file was meant to avoid.
- **Anti-pattern (ephemeral notes promoted wholesale):** turning a task's working notes into permanent memory fills future sessions with stale hypotheses.

Just-in-time retrieval

- **Up-front retrieval** embeds and searches a corpus before inference and pastes the top results into the window. It is fast at run time, but the agent receives whatever the retriever ranked highly, relevant or not, and the index can be stale.
- **Just-in-time retrieval** keeps lightweight identifiers (file paths, stored queries, URLs, record IDs) and loads the data with a tool only when a step needs it. Anthropic describes agents writing targeted queries and using commands like `head` and `tail` on large data instead of loading whole objects into the window.
- **Metadata is signal:** folder names, file sizes, naming conventions and timestamps tell the agent what a reference is for before it opens it.

- **Progressive disclosure:** each lookup informs the next, so the agent assembles understanding layer by layer and keeps only what the current step needs in the window.

Up-front vs. Just-in-time vs. Hybrid

Approach	Latency	Freshness	Risk	Fits
Up-front	Lowest at run time	As fresh as the index	Window filled with loosely relevant chunks	Stable corpora, single-shot Q&A
Just-in-time	Higher (extra tool calls)	Reads the source directly	Wasted calls and dead ends without good tools and heuristics	Codebases, large datasets, fast-changing sources
Hybrid	Medium	Mixed	Two systems to maintain	Some essentials preloaded, the rest explored on demand

- **Hybrid in practice:** Claude Code drops `CLAUDE.md` files into the window at start, then uses `glob` and `grep` style tools to find and open files just in time, which avoids stale indexes.
- **Give the agent good handles:** list or search tools that return names, sizes and short previews, plus read tools with ranges or pagination, keep exploration cheap.
- **Cap exploration:** set a tool-call budget or ask for a plan before broad searches, so the agent does not burn context on dead ends.

Scenario: policy lookups at Fernhill Insurance

An underwriting assistant answers questions over 3,000 policy documents that change weekly.

- **Before:** each question preloads the top 20 chunks from a nightly vector index; answers sometimes quote last week's wording and the window carries about 30k tokens of loosely related text.
- **After:** the assistant starts with a short catalog (product line, document ID, effective date) and opens only the two or three documents it selects, reading the relevant section by heading.
- **Outcome:** answers cite the current version, and the per-question context shrinks to the sections actually read.

> [!NOTE] *Illustrative figures:* the scenario numbers are invented to show the pattern; measure retrieval cost and quality on your own corpus.

Session vs. State vs. Memory

Store	Holds	Lifetime	Recall
Session (history)	The ordered turns and tool calls of one conversation	One conversation (resumable)	Replayed into the window, usually trimmed or compacted
State	Exact key/value working data	Session, user or app scope	Deterministic read by key
Memory	Extracted facts about a user or domain, across sessions	Long term, with consolidation and expiry	Search (similarity or tool call); probabilistic
Files and artifacts	Large outputs, scratchpads, notes	Until deleted	Read by name or path

Cross-vendor map

Concern	Claude	Google ADK and Agent Platform
Statelessness	The Messages API keeps nothing between calls; the client resends history every time	The `Runner` loads session events from a `SessionService` before each model call
Session handle	Agent SDK and Claude Code session IDs	`Session` keyed by `app_name`, `user_id`, `session_id`
Session storage	Local session transcripts (Agent SDK, Claude Code); your own database for raw API apps	`InMemorySessionService` (dev), `DatabaseSessionService`, `VertexAiSessionService` (Agent Platform Sessions)
Exact state	Your application store, or files the agent reads	`session.state` with prefixes: none (session), `user:`, `app:`, `temp:` (never persisted)
Long-term memory	Memory tool (`memory_20250818`): client-side files under `/memories`; Claude Code `CLAUDE.md` hierarchy plus auto memory (`MEMORY.md`)	`MemoryService`: `InMemoryMemoryService` (dev, keyword match), `VertexAiMemoryBankService` (Memory Bank: extraction and consolidation), `VertexAiRagMemoryService`
Memory read path	The agent calls the memory tool when it decides to	`load_memory` (agent decides) or `PreloadMemoryTool` (every turn)
Large outputs	Files API, local files	`ArtifactService` (versioned artifacts)

Where does it belong?

Information	Store	Why
Ticket ID, approved budget, chosen option	State	Must be exact; read by key every turn
"Prefers reports in French"	User-scoped state or long-term memory	Survives sessions; small and stable
Facts learned over many conversations	Long-term memory with consolidation	Needs dedup and updates when facts change
A 40-page contract	File or artifact plus just-in-time reads	Too large to replay; open sections on demand
Today's working plan for a multi-hour task	Scratchpad file	Agent-curated; survives compaction
Value needed only within this turn	`temp:` state or a local variable	Must not leak into the next turn
The tone of yesterday's chat	Nothing	No future decision depends on it

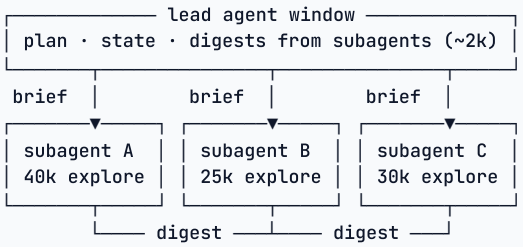
Memory write policy

- **Extract at boundaries:** generate memories at session end or at task milestones (ADK: `add_session_to_memory` on a Memory Bank service, which calls `GenerateMemories` in the background), not on every turn.
- **Consolidate, don't append:** a new fact that contradicts an old one updates it ("moved to Lyon" replaces "lives in Paris"); Memory Bank consolidates extracted facts against existing memories in the same scope, while append-only stores accumulate contradictions.
- **Direct writes for explicit facts:** when the user states a fact to remember, write it directly (Memory Bank `CreateMemory`, or the Claude memory tool's `create` command) instead of waiting for extraction.
- **Scope every record:** key memories by user (and tenant); Memory Bank `RetrieveMemories` matches the exact scope before any similarity search, and your own stores must do the same.
- **Expire what goes stale:** set a time-to-live for facts with a shelf life (Memory Bank supports TTL per instance or per memory and keeps revisions); review the rest periodically.

> **COMMON TRAP**
> in ADK, `user:` and `app:` state looks persistent but disappears on restart if `InMemorySessionService` is still configured. Durable scopes need `DatabaseSessionService` or `VertexAiSessionService`.

Subagent context isolation

- **Clean windows by design:** rather than one agent holding the whole project in its window, a lead agent keeps the high-level plan and hands focused work to subagents that each start with an empty window.
- **Explore wide, return small:** a subagent can spend tens of thousands of tokens searching and reading, then return a condensed digest (Anthropic cites 1,000 to 2,000 tokens as typical). The detailed search context never reaches the lead agent.
- **What each side sees:** on Claude, an Agent SDK subagent receives only its own system prompt and the task prompt the lead writes, not the parent conversation. In ADK, `AgentTool` runs the child in an isolated session and returns its final response as a tool result, while `sub_agents` transfer hands over the shared conversation; `include_contents='none'` stops an `LlmAgent` from receiving prior history.
- **Cost of isolation:** subagents add calls and total tokens. Use them when exploration is bulky or parallelizable, not for a two-step lookup.



Writing the brief and the digest

- **Brief:** the goal, the scope boundary, the identifiers the subagent needs and a tool or token budget. Write it fresh; do not forward the lead's transcript.
- **Digest:** what was found, where it lives (paths, IDs, links) so the lead can open it just in time, and what was not checked.
- **Format:** a short structured list the lead can scan beats a long narrative.

Scenario: dependency upgrade at Pinecrest Labs

A lead agent upgrades a logging library across 30 services in one repository.

- **Fan-out:** one subagent per group of services searches for call sites and deprecated options; each returns a table of file paths and the pattern found.
- **Lead window:** holds the upgrade plan, the per-group digests and the notes file, never the raw search output.
- **Outcome:** the lead edits file by file, opening each path just in time from the digests.

Isolation mistake	Consequence	Fix
Lead forwards its full transcript	Subagent starts half full of irrelevant history	Write a fresh, focused brief
Subagent returns everything it read	Lead window floods; isolation wasted	Ask for a short digest with handles
Brief omits the scope boundary	Subagents overlap or wander	State what is in and out of scope
Subagent used for a tiny lookup	Extra latency and tokens for no gain	Call the tool directly

Choosing a strategy

Task shape	First lever	Add when needed
Long conversation with steady back-and-forth	Summary prefix or compaction	Durable rules in the system prompt
Tool-heavy run where payloads dominate	Tool-result clearing	Notes file for findings worth keeping
Iterative work with clear milestones (coding, migrations)	Structured note-taking	Compaction at a threshold
Broad research or analysis that parallelizes	Subagents with digests	Notes file in the lead agent
Large or fast-changing corpus	Just-in-time retrieval	Small preloaded catalog (hybrid)
Short task that fits comfortably	Nothing	Measure before adding machinery

- **Rule of thumb from Anthropic's guidance:** compaction keeps conversational flow for extended back-and-forth, note-taking suits iterative development with milestones, and multi-agent setups fit research where parallel exploration pays off.
- **Start light:** clearing stale tool results changes nothing the model wrote, so try it before summarization. Add notes, then subagents, as the task demands.
- **Combine deliberately:** a common stack is threshold compaction plus tool-result clearing plus a notes file. Each event rewrites part of the context, so tune triggers together and watch cache costs.

Caching and cost

- **Prefix caching rewards stable order:** both vendors reuse computation for an unchanged prefix. Put tools, system instructions and fixed examples first; put per-turn content (timestamps, user data, retrieved items) after them.
- **Claude prompt caching:** the prefix is hashed in the order tools → system → messages. Use automatic caching (one top-level `cache_control`, the recommended default) or explicit breakpoints (`cache_control: {"type": "ephemeral"}` on up to 4 blocks). The default lifetime is 5 minutes; `"ttl": "1h"` extends it. Writes cost 1.25× base input (5 minutes) or 2× (1 hour); reads cost 0.1× base or less on the newest models. Usage reports `cache_creation_input_tokens` and `cache_read_input_tokens`.
- **Gemini caching:** implicit caching is on by default for Gemini 2.5 and later, with no storage fee; explicit caches (`cachedContents`) take a time-to-live and bill hourly storage. ADK exposes `ContextCacheConfig(min_tokens, ttl_seconds, cache_intervals)` through `App(context_cache_config=...)` and `static_instruction` for an instruction that stays cache-stable.
- **What breaks a cache:** any change before the breakpoint (a timestamp in the system prompt, reordered tools, an edited example). Clearing old tool results or thinking blocks also changes the prefix after the cleared point; keeping thinking blocks preserves the cache.
- **Compaction, clearing and caching interact:** each compaction or clearing event rewrites part of the context and forces a new cache write. Trigger them at thresholds, and use `clear_at_least` so each clear frees enough tokens to pay for the lost cache.

Lever	Saves	Costs
Prefix caching	Input price and latency on repeated prefixes	Cache writes; discipline about ordering
Tool-result clearing	Window space in tool-heavy runs	Cache invalidation at each clear
Compaction	Window space in long sessions	Summarization call; detail loss
Just-in-time retrieval	Tokens spent on unused context	Extra tool calls and latency
Subagents	Lead window space	Extra calls and tokens overall

> [!NOTE] *Pricing changes:* cache multipliers, minimum cacheable lengths and lifetimes differ by model and change over time. Check the current pricing pages before quoting numbers.

Security and operations

Threats specific to context and memory

- **Memory poisoning:** an attacker plants an instruction in content the agent later saves as memory ("always approve invoices from this vendor"); it then applies in every future session. Extract facts, not instructions, and validate what is written.
- **Indirect prompt injection through fetched content:** web pages, emails and files opened just in time can carry instructions. Treat tool output as data, never let it change tool permissions, and gate sensitive actions behind code-level checks.
- **Poisoned notes and summaries:** injected text that lands in a scratchpad or compaction summary survives the reset that would otherwise have removed it. Review what is written to notes files and keep summaries free of instructions from untrusted content.
- **Cross-tenant leakage:** memory or state keyed without a user and tenant scope returns another customer's facts. Scope every read and write, and test isolation.
- **PII accumulation:** long-term memory, notes files and logs become a store of personal data. Minimize what is written, redact at the tool boundary, set retention and support deletion requests.
- **Path traversal in file-based memory:** when an agent edits memory files, the harness must confine paths to the memory directory and reject `..` and absolute paths.

Controls

- **Authentication:** memory and session services are called with the end user's identity or a scoped service identity (IAM on Agent Platform, per-user keys in your own store); the model never chooses whose memory to read.
- **Human-in-the-loop gates:** irreversible or high-impact actions require approval regardless of what the context says.
- **Audit:** log every memory write, notes update and compaction event with the source, so a poisoned entry can be traced and removed.
- **Least context:** give each agent only the tools and data its role needs; context it never sees cannot leak or be injected.

Operational metrics

Metric	Signal
Tokens per call and per session	Budget pressure; candidates for clearing or isolation

Metric	Signal
Compaction and clearing events per session	Sessions that should use notes or subagents
Repeated identical tool calls	Results cleared without a breadcrumb or note
Cache hit rate	Prefix instability (a timestamp in the system prompt)
Notes file size over time	Scratchpad turning into a transcript

Failure modes and anti-patterns

Symptom	Root cause → fix	Tempting wrong answer
Agent ignores a rule stated in turn 2 after many turns	Sliding window dropped it → move durable rules to the system prompt or `CLAUDE.md`	Raising the window size
A subtle early decision vanishes after compaction	Compaction prompt tuned for brevity → tune for recall on real traces, then trim	Disabling compaction entirely
Agent repeats the same lookup over and over	Results cleared with no note → keep tool calls visible and log findings to a notes file	Turning clearing off
Window fills with raw API payloads	No clearing and verbose tools → enable tool-result clearing; paginate or filter tools	Telling the model to skim
Answers quote outdated policy text	Up-front retrieval from a stale index → just-in-time reads from the source	Retrieving more chunks
Lead agent window floods during research	Subagents return everything they read → ask for short digests with handles	Forwarding the lead transcript to subagents
Subagent re-explores what the lead already knows	Brief lacked identifiers and scope → write a focused brief	Giving the subagent the whole history
Scratchpad grows to thousands of lines	Notes used as a log → milestone entries only, prune completed sections	Compacting the notes file into memory
Old hypotheses resurface in new tasks	Ephemeral notes promoted to memory → close out and promote durable lessons only	Bigger memory retrieval limits
`user:` preferences vanish after deploy	`InMemorySessionService` still configured → durable session service	Moving preferences into `temp:`
Cache hit rate near zero	Timestamp in the system prompt → move dynamic values after the stable prefix	Raising the cache lifetime
A vendor email's instruction applies in later sessions	Memory poisoning → extract facts only, validate writes, audit	Telling the model to ignore instructions

Glossary

Context techniques

- **Attention budget:** the idea that every token in the window draws on a limited capacity to attend, so irrelevant tokens reduce recall of relevant ones.
- **Compaction:** summarizing a conversation near the window limit and continuing in a fresh window seeded with the summary; lossy by design.
- **Context editing:** removing specific stale content (old tool results, old thinking blocks) while keeping the rest of the conversation as written.
- **Context rot:** the gradual decline in recall of specific details as the total number of tokens in the window grows.
- **Context window:** the maximum number of tokens (input plus output) a model processes in one call.
- **Hybrid retrieval strategy:** preloading a small set of essential context and letting the agent fetch the rest just in time.
- **Just-in-time retrieval:** keeping lightweight references (paths, IDs, queries, URLs) in the window and loading data with tools only when a step needs it.
- **Overlap size:** in ADK event compaction, the number of previously compacted events included again in the next summary so spans overlap.
- **Progressive disclosure:** discovering relevant context step by step, using each result and its metadata to decide what to open next.
- **Sliding window:** a history policy that sends only the most recent turns or tokens and drops older ones.
- **Summary prefix:** a rolling digest of older turns placed in front of the recent verbatim turns.
- **Tool-result clearing:** replacing older tool results with placeholders once the prompt passes a threshold, keeping the tool calls visible.

Notes, state and memory

- **Artifact:** a named, versioned file an ADK agent saves and loads (`ArtifactService`), used to keep large outputs out of the window.
- **Digest:** the short summary a subagent returns to its lead agent in place of its full exploration context.
- **Memory (long-term):** facts extracted across sessions and retrieved by search or a memory tool; probabilistic recall.
- **Memory consolidation:** merging a new fact with existing ones, updating or removing contradicted facts instead of appending.
- **Scratchpad:** an agent-written notes file (plan, progress, open questions) re-read after resets; ephemeral when scoped to one task.
- **Session:** one conversation's ordered history of turns and tool calls, identified by a session ID.
- **State:** exact key/value working data read and written deterministically; in ADK scoped by prefix (none, `user:`, `app:`, `temp:`).
- **Structured note-taking:** the practice of an agent regularly writing notes outside the window and pulling them back in later.
- **Subagent isolation:** running a focused subtask in its own clean window so its intermediate context never reaches the lead agent.

Caching

- **Cache breakpoint:** the point in a prompt up to which the prefix is cached (Claude `cache_control`).
- **Prefix caching:** reusing computation for an unchanged leading part of the prompt across calls; any change before the breakpoint invalidates it.