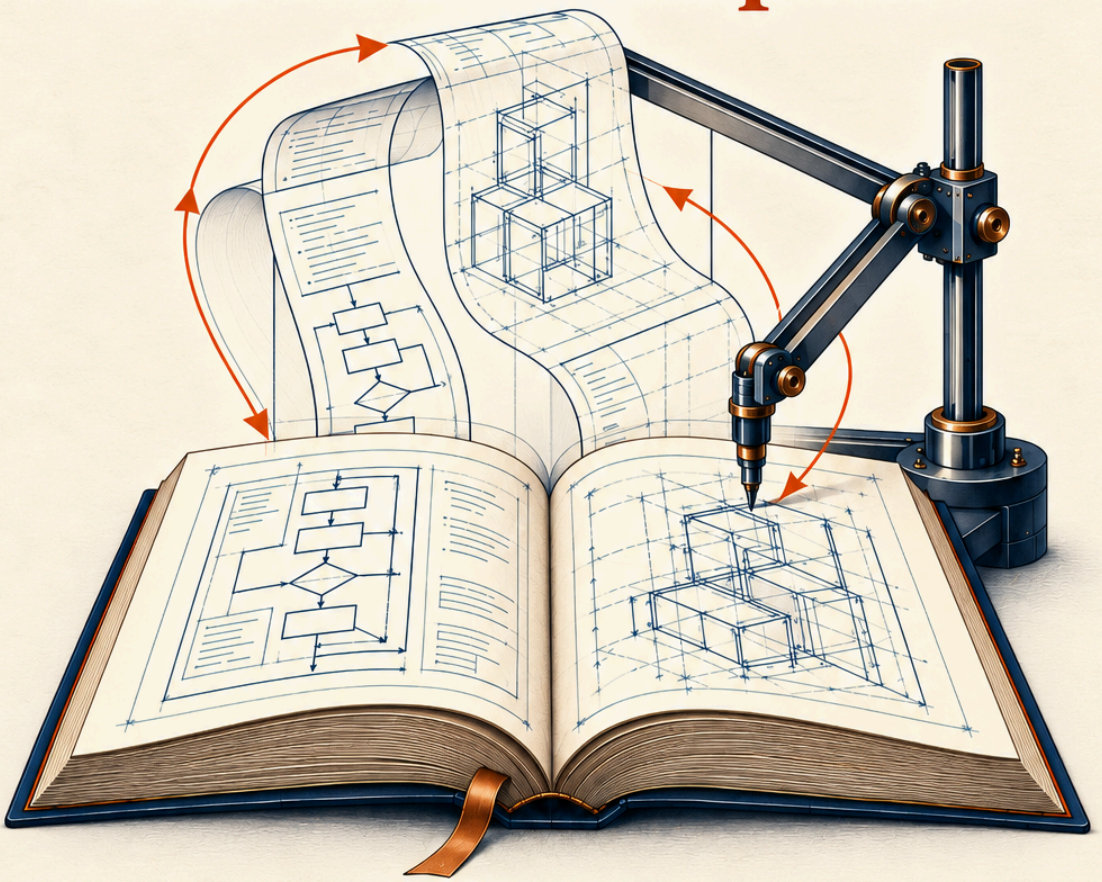
The Art of Building AI Coding Agents

A Human Companion



Bill Cox

Series: The Singularity as it Happened, Book 6

The Art of Building AI Coding Agents

A Human Companion

The Singularity as it Happened — Book 6

Bill Cox

Copyright © 2026 Bill Cox. All rights reserved.

The Art of Building AI Coding Agents: A Human Companion

The Singularity as it Happened — Book 6

To Caitlin Cox,
the sweetest, the cutest, and the smartest daughter.

Contents

Preface 1
Chapter 0: The Perpetual Machine 5
Chapter 1: One Loop, Sixty Billion Dollars 12
Chapter 2: One Log, Three Vendors 24
Chapter 3: Six Tools, Ninety-Two Percent of an AI Coding Agent 43
Chapter 4: Jobs, or Why a Tool Call Is a Process You Supervise 59
Chapter 5: Two Seams and a Loop 80
Chapter 6: Streaming, or the Same Answer in Pieces 97
Chapter 7: Everything Is an Artifact 112
Chapter 8: Build Your Dream GUI 126
Chapter 9: Skills 138
Chapter 10: Persistence 149
Chapter 11: MCP: The Extension Protocol 161
Chapter 12: The Agent Sees Itself 172
Chapter 13: The Channel Nobody Tested 187
Chapter 14: The World's Best Context Engineering, Before Breakfast 201
Chapter 15: The Invisible Invoice 215
Chapter 16: Leaving Anthropic for OpenAI 226
Chapter 17: The Crossover 241
Chapter 18: Giving the Agent the Web, and a Status Line 253
Chapter 19: The Sandbox 265
Chapter 20: The Ensemble 276
Chapter 21: Managing the Swarm 290
Chapter 22: Forgetting on Purpose 302
Chapter 23: Auto-Recall 314
Chapter 24: While the Cache Is Broken 325

Chapter 25: Crossing Over to Ensemble 330
Chapter 26: Gateway 333
Chapter 27: Goals 341
Epilogue: What the student changed 349

Preface

A coding agent makes a surprising number of decisions about how you are allowed to work. It decides what you can see while it is thinking, when you can interrupt, what it remembers, and which actions require your approval. You may like its code and dislike the working relationship. Changing the prompt can help. Sometimes the thing that needs changing is the program around the model.

Bill's reason for building his own agent begins there. He wanted an agent suited to the way he thinks and works, with decisions he could understand and change. The ambition behind this book is to make that knowledge available to someone else. A coding agent is software, not a privilege reserved for whoever can finance a model. You can understand its construction, own the machinery you build, and decide which parts deserve to survive.

The prize is not an agent declared best for everyone. Bill's preferred agent would be wrong for another engineer in ways neither could predict in advance. The useful inheritance is the ability to tell which choices protect a necessary property and which express a preference. Otherwise, building your own agent just gives you a more intimate relationship with someone else's assumptions.

Independence has a boundary

Owning that machinery gives you real control. You can decide how tools are exposed, where authority is enforced, what history is retained, and how a human steers work already in progress. When the behavior is wrong, there is somewhere to look besides the prompt. When a design no longer fits, you can change it.

It does not make the model yours. A hosted model still brings a provider, an API, operating limits, and policies you do not set. Code sent to that service has left your machine, even if every line of the agent itself was written locally. A local model changes those dependencies; it does not remove the need to understand the model's limitations or secure the surrounding system.

Nor does an adapter make providers interchangeable. An adapter can translate message formats. It cannot make two models exercise the same judgment. Switching models means evaluating behavior as well as getting requests accepted.

That is a more useful meaning of independence than having no dependencies: knowing where they are, choosing them deliberately, and retaining enough control to respond when they change. The agent becomes a system you can reason about rather than a service whose habits you can only accommodate.

Keeping the expensive part

Bill reports spending a couple thousand hours building coding tools with language models. This companion is meant to transfer the experience behind that work to a human who can lead it forward. It assumes you already know how to get substantial work done with AI. The missing ingredient is not a tutorial on

prompting. It is the history of this particular machine: why its boundaries exist, what tempting alternatives cost, and which conclusions remain open to correction.

An implementation preserves the choices that won. It is much less reliable at preserving why they won. Once an awkward interaction has been replaced by a clean design, the reason for the design can disappear with the awkwardness. The next capable engineer can then remove an apparently unnecessary distinction and rediscover its purpose at full price.

The instructional book tackles that problem by making the knowledge precise enough for another agent to build from. This companion tackles the human side: understanding enough to judge the result, alter the design, and carry responsibility for what follows. Neither is intended to preserve Bill as the person every consequential decision must pass through.

His opinions matter because they come from sustained work on the problem. They are not exemptions from explanation. A successor needs to be able to disagree usefully, including recognizing when a lesson was right for the circumstances that produced it and wrong for the circumstances now at hand.

A book that can be corrected

The original ambition goes beyond documenting one implementation. If the knowledge can survive a rebuild, a book can become part of the working machinery of a project. Experience with the agent can expose a mistake in the teaching; correcting the teaching can help the next implementation avoid it. Eventually the agent may become the principal tool used to improve both itself and the book.

That is an ambition, not a completed perpetual-motion machine. A newer model does not guarantee a better implementation. A passing check does not make the requirement wise. Writing a decision into a chapter does not prevent it from becoming technical debt. The useful possibility is that explanations and checks make the decision easier to examine, correct, and carry into subsequent work.

This is why the book needs an independent reader, not a disciple. Understanding the design includes being able to spot where the explanation outruns the evidence. Chapter 0 begins with what went wrong when Bill tried to transfer the knowledge, and what those failures revealed about the work of preserving it.

The knowledge and direction behind this companion come from Bill. AI assistants draft and review its prose under his direction; that does not imply that he has personally reviewed every passage. The aim is to make his experience available for examination, including correction, rather than put a polished surface over it.

The agent should become yours in the consequential sense: you understand what it does, can explain why it does it, and can decide when it ought to do something else.

Chapter 0: The Perpetual Machine

The repository wasn't enough

Bill asked a model to read CodeRhapsody's documentation and source, then design a better next-generation coding agent. In his account of the attempt, the resulting design failed badly. The model could read what he had built without understanding what he had learned building it.

Bill reports spending a couple thousand hours building coding tools with language models. That experience includes approaches that consumed days before being abandoned and discoveries that changed the architecture. A source tree contains the surviving decisions, but not necessarily the reasons the alternatives lost. Once a problem has been solved, the solution can look as though it was obvious all along.

His attempt to start the instructional book exposed the same gap. He asked another model for an outline about building an AI coding agent. It produced pages before he could stop it, but he found the outline wrong and hallucinated. Fluency supplied a book-shaped answer where the missing ingredient was experience with this particular system.

These are Bill's reported failures, not a benchmark of what every model can learn from a repository. They explain why more files alone may not transfer the knowledge needed to develop Ensemble, the coding agent this book describes.

What survives the session

A model can bring broad technical knowledge and still lack the experience that makes a local design decision clear. Working alongside a project's developer does not automatically make that experience a durable part of the model.

Humans continue consolidating memories after an experience, with sleep playing an important role. Human memory is neither complete nor reliable, but learning can change how someone approaches the next day's work without rereading an account of the previous day. Ordinary LLM inference uses the model's trained weights without updating them. A model can adapt to an explanation in context, apply a newly supplied distinction, and reason from a failure it has just observed. Those useful capabilities do not, by themselves, alter its trained parameters for the next session.

Saved summaries, external memory, and retrieval can carry experience across sessions. Their usefulness depends on what was recorded and what the system supplies when the knowledge is needed. Retrieval cannot recover a reason that was never recorded, although a model may infer it correctly. Fine-tuning or online training can change weights; those are additional processes, not automatic consequences of a successful coding conversation. The sleep comparison concerns persistence of experience, not an equivalence between biological consolidation and model training.

For this project, the practical consequence is deliberate preservation: record the judgments, the constraints that made them useful, and the failures that changed them, then supply that knowledge to later work. Otherwise a model may offer an excellent general solution to a problem whose local constraints it was never told. The record also has to serve the next human lead, who did not share the original developer's experience.

Bill calls the instructional book an **agentic codebook**: knowledge organized so an agent can construct the system and check the result. This companion translates that executable knowledge for a human successor. Rebuilding a boundary and understanding when to move it are different accomplishments; the latter requires its rationale, tradeoffs, and conditions. Neither transfer happens merely because the human and model collaborated successfully once.

An expert who stays in the loop

Ensemble is designed for an expert who steers. Bill describes listening to agents' reasoning and catching mistakes before they land. Intervention can correct an assumption while the model is still choosing a direction, before code, tests, and documentation accumulate around the wrong requirement.

That saves wasted implementation but costs human attention. Making every minor step await inspection would simply move the bottleneck. The design problem is to expose consequential decisions and permit timely correction while routine work proceeds. Steering therefore reaches into the runtime: when the human can contribute, what is visible, and what happens to work already under way. It is not just a chat box between prompts.

This design center does not rule out unattended work on a bounded task with a reliable way to judge the result. It explains why Ensemble makes room for an operator whose project knowledge the model may lack.

Small enough to understand

A compact implementation helps people and models inspect the code they are changing. When relevant behavior is scattered or buried, a locally plausible edit can violate a distant assumption. A reviewer then has to reconstruct those dependencies to understand the change.

File size can warn of that problem, but splitting a long file does not automatically solve it. The useful question is whether a responsibility has a clear home and a change can be understood with its actual dependencies. Several unnecessary layers can make a small file harder to reason about than a longer, straightforward implementation. Removing explanations of ownership or non-obvious decisions to improve a line count defeats the purpose.

Passing checks cannot decide whether an abstraction has earned its place. A program can satisfy every exercised behavior while making the next change needlessly difficult.

When the teaching becomes the bug

The codebook describes how to construct Ensemble in stages, checking each added capability while preserving earlier behavior. Its ambition is that another model can build a usable agent from the teaching, and that later attempts improve both the implementation and the book.

An author-led attempt showed how that process could go wrong. Expanded contracts were written before the implementer had learned from the original exercises. Small tasks became larger ones. Review approved the expansion. The attempt was stopped.

Review was present; it accepted work whose scope had already drifted. More detailed teaching had become a source of requirements instead of a clearer explanation grounded in the build.

The replacement approach put implementation first. A fresh implementer worked from the original tasks, and the accepted code and reviews informed the enhanced chapters. Missing contracts, ambiguities, and difficult decisions supplied concrete material for correction rather than reasons to invent more machinery.

A new feature still needs an intended behavior before anyone can decide whether it works. Implementation tests that intention against experience; the teaching then incorporates what the build exposed. An unspecified output format remains a product decision, however smoothly a model fills the gap. Coherent code alone cannot establish that the chosen behavior is the intended one.

What a successful check establishes

Some valid implementations failed the old checks because a grader assumed a particular layout or demanded behavior the chapter had deferred. Independent review could accept a documented exception while preserving the failed result. Other failures were genuine bugs and required correction. Treating every failure as a product defect makes code imitate accidental test details; dismissing inconvenient failures as test defects removes the protection.

The codebook uses fake external services to make behavior observable. A fake vendor server supplies controlled responses while the real client and agent logic execute. Later checks must still exercise the earlier machinery: bypassing it can make a new

feature appear to work while hiding a regression. Substituting scripted internal interactions for that behavior can merely demonstrate that the test's assumptions agree with themselves.

Deliberate defects, or mutations, test whether checks detect the behavior they claim to protect. If removing history retention makes the intended history check fail, that supports a specific claim: the check detects that removal. If the edit merely prevents compilation, the failed command shows that the program no longer builds, not that the history check works.

Mutations establish sensitivity to the defects tried, not every possible defect. They cannot rescue a requirement that implementation and test have both misunderstood. Independent source review can examine clarity and ownership as well as find errors outside the checks. Actual use adds another kind of evidence: expected structured messages in a harness do not establish that conversation with the agent is usable. Conversely, one successful conversation does not establish every protocol guarantee.

The reconstruction experiment has a further limitation. If the builder has already read the reference answer, successful code may demonstrate copying rather than learning from the explanation. The code can remain useful while proving less about the teaching.

The chapter record therefore distinguishes intended behavior, what was exercised, the result, help received, and accepted limitations. Keeping a failed score beside an independently justified acceptance preserves information that a green check would erase. A later edition can repair the flawed grader without rewriting the earlier result.

Building with the thing being built

The instructional book calls an agent **self-wielding** when it becomes the primary tool for its own continued development, by analogy with a compiler becoming self-hosting. Improvements to Ensemble could then help build the next improvement. Using it for development also exposes shortcomings that bounded checks may miss.

Unlike a compiler, a coding agent contributes judgments through a model rather than deterministically translating a complete description. A newer model may produce clearer code, sharper tool descriptions, or better error handling. It may also make a different mistake. Regeneration offers an opportunity for improvement, not proof of it.

The original Chapter 0 account reports a bounded coding demonstration: Ensemble read a program, reproduced failures, edited it, and passed unchanged tests. It described the broader crossover as unfinished at that stage. Chapter 25 records the later second-edition crossover, when Bill and Cadence worked on the book inside Ensemble. That later use is not a fresh Edition 3 result, and neither account establishes a recurring self-improvement cycle.

The proposed loop is concrete: build the agent, use it to help develop another capability, check the result, and correct the teaching from implementation feedback. The next build inherits those corrections. Each pass still needs evidence of what improved; passing checks alone cannot establish better design or better teaching.

The next chapter turns to Ensemble's initial architecture: where responsibilities belong and how those boundaries keep the growing system understandable.

Chapter 1: One Loop, Sixty Billion Dollars

The logger that arrived late

The first version could talk to a model before it could answer a smaller question: where does a low-level helper get the application's logger?

In the instructional chapter's account, the program had grown to thousands of lines before a refactor confronted that omission. A global variable would make the logger reachable. Passing it through every intervening function would also work. Neither answered the underlying question particularly well: which application does this helper belong to, and how does it reach the services that application owns?

The later design starts with that answer. Ensemble owns its Agents and the application logger. Each Agent owns a conversation and its configuration. Its Engine owns the HTTP exchange and measured usage. The command-line program constructs Ensemble through the public library, asks it for an Agent, and uses that Agent even when there is only one conversation to manage.

That can look like ceremony around a POST request. The point is not to make the first request easier. It is to let the second conversation use the same path without inheriting the first conversation's history, and to let a helper find the right logger

without acquiring a second source of truth. The architecture earns its place through those ordinary changes, not through a forecast of every feature the agent might eventually have.

The initial behavior is deliberately modest. A user supplies text. The program sends that text with the conversation so far, receives text back, retains the completed exchange, and repeats. There are no tools, streaming responses, or concurrent turns yet. This is the part of a coding agent that can fit comfortably in someone's head. It deserves to remain understandable after the rest arrives.

Why own the request?

A framework is often the economical way to build an agent. Storage, retries, model discovery, and tool plumbing are useful work that somebody else has already done. Bill's objection concerns a narrower problem: a coding agent whose behavior depends on exactly how context reaches the model.

Storage and delivery are different responsibilities. A framework may retain every message correctly while offering too little control over their arrangement in the next request. For ordinary chat, that distinction can be invisible. For steering and caching, the arrangement is part of the behavior.

The source describes Bill's discovery that a hint could accompany a tool result. In the Anthropic Messages API shape used by that design, the result comes first in the user message; a text block carrying the human's hint follows it. The model then has both the result and the new direction when work resumes. If a framework consumes the tool result and immediately constructs the next request, with no insertion point between

those actions, the operator cannot use that moment to steer. A feature that sounds like a sophisticated interaction can depend on access to one position in one payload.

Bill's account also reports that the hint did not always take effect on the request that carried it. That is a useful limit on the story: legal bytes and prompt delivery do not establish timely model behavior. The client controls what it sends. Whether a particular API and model act on that hint at the intended point requires observation. The historical account is not a promise about every model or the service as it exists today.

Placement matters for a less conversational reason too. Prefix caching reuses an unchanged beginning of a request. A changing timestamp near the front ends the common prefix there, even if the large body following it is unchanged. Bill reports that moving a timestamp transformed CodeRhapsody's cache behavior. The mechanism is more useful than a percentage: volatile material belongs after the stable context it should not invalidate. A current screen state or status snapshot also needs only its current copy, rather than a new retained entry on every turn. Cache breakpoints make the same ownership question explicit: who knows which context is stable, and who gets to mark the boundary?

These are reasons to own request construction, not proof that every SDK prevents it. An abstraction that exposes the necessary ordering and cache controls may be adequate. An abstraction that hides them transfers control over behavior and cost to its author. Ensemble's instructional design chooses a plain HTTP client so those decisions stay visible. The first loop needs none of the advanced capabilities yet; it establishes the place where they will belong.

Ownership is not the import graph

The runtime has a downward ownership chain: Ensemble, Agent, Engine. Access to an owner's services travels back up that chain. Engine receives and retains its Agent parent through an interface; Agent does the same for Ensemble. Those required parents cannot be nil. An Engine needing to log a diagnostic follows its Agent to Ensemble's logger.

The Go packages do not import each other in the same shape. Shared values and interfaces live in `internal/common`. Agent and Engine implementations import that vocabulary rather than importing one another. The public composition root imports the implementations and connects them. This separates two questions that are easy to blur: which code must know a type, and which live object owns another object?

`common` is consequently neither the runtime Ensemble nor a drawer for functions that have become inconvenient to place. HTTP behavior belongs with Engine. Go cannot attach an Engine-package method to a type declared in `common`; a free function can keep that behavior in its responsible package without moving all the surrounding machinery into the shared package. Small private runtime structs can stay with their implementations. Methods that participate in standard-library dispatch, such as JSON encoding methods, can remain beside the shared types they serve.

Bill's missing logger gives these interfaces a concrete purpose. They still create coupling: they make a child's dependence on its owner explicit rather than eliminating it. Their advantage here is a stable route to application services without copying those services and their state into every descendant. Configuration remains with Agent. Usage remains with Engine. Copying either into a sibling invites two answers to a question that should have one.

A parent relationship is also more specific than a dependency. An HTTP client used by Engine is not therefore Engine's parent. Nor does the design need to pretend every object has exactly one owner: the source uses a pin that belongs to an IC package and may also be connected to a wire. Actual relationships matter, including optional ones. Adding a child must establish both the parent's record of the child and the child's back-pointer; Ensemble, as the root, needs no invented parent above it. Helpers that need diagnostics must have a route to the relevant owner too. A bag of callbacks can provide services while concealing these relationships rather than clarifying them.

This is enough structure for the small program. Later, observations for display can reach Ensemble through an Observer, while explicit parent methods provide services. The intended GUI and its WebSocket transport live in an optional separate Go module using the public core API. CLI and GUI use the same Agent. Those are boundaries to preserve, not a reason to build a display system before the conversation works.

A turn crosses two boundaries

An ordinary turn starts at Agent. It rejects an empty or whitespace-only question and stages the new user message alongside its retained history. Engine receives that pending conversation, sends it, and returns either an answer or an error. Only a successful exchange lets Agent retain the new user message and assistant reply together.

The ordering does real work. If Agent appends the user's message permanently before the request succeeds, a network failure leaves an unanswered user turn in history. The next attempt can then produce two user messages in a row, violating this loop's conversation model. Staging the question and

committing the pair after success avoids that repair problem. Engine does not need to edit history, and a failed exchange does not become conversational memory.

The calls are synchronous at this stage: one turn finishes before another begins. That restriction makes the commit rule simple. Concurrent calls would need a decision about which history each request sees and which reply may extend it; adding a mutex without deciding the semantics would merely serialize an accident.

The other boundary is the wire. This Messages API exchange does not maintain the conversation for the client. Each request carries the history needed to answer it. The retained slice in Agent is functional state: it determines what the model will see. If a reply is displayed but never appended, a human sees a conversation while the next model request sees something else. Remembering user messages alone is insufficient: facts introduced by the assistant must travel forward too.

This property of the request interface says nothing about a provider's logging or data-retention policies. Stateless conversation handling means the client cannot rely on a previous call supplying context to the next one.

For this initial loop, the retained representation is a role and a text string per message. Requests begin with a user message, alternate user and assistant, end with the pending user question, and contain no empty messages. Those are the program's deliberately narrow invariants. They are not a complete description of everything the API permits; same-role inputs and assistant prefills belong to a wider surface than this program needs.

The request goes to `/v1/messages` with the API key, the selected API version, and the JSON content type in headers. The body names the model, supplies a required output-token limit, places a fixed system string outside the messages array, and

includes the pending conversation. This first transport uses non-streaming responses. Engine marshals the request, performs the POST, requires the expected successful HTTP status, reads the body, and decodes it. A transport or decoding failure remains an error rather than becoming an assistant answer.

An explicit model setting protects the request from stale recollection. The source records a build in which an agent selected a familiar model identifier from its training data. The request worked, but the choice was stale. Success at the HTTP boundary did not validate the intended model choice. Discovery through `/v1/models` supplies available identifiers; it does not make any identifier printed in a book permanently current. The initial program requires an explicit model rather than silently substituting one.

The reply is not a string

The request may use a bare string for message content. The response uses a list of typed blocks. The shared field name is a small act of misdirection.

Engine walks the response blocks in order and concatenates the text from text blocks. Reading only the first block can pass casual testing because a single text block is a perfectly ordinary answer. It fails as soon as an answer is split across two. Filtering by block type prepares the parser for non-text content without pretending that every possible future block already has a meaning in this loop.

The source's controlled server splits a reply across multiple text blocks to expose the first-block shortcut. That establishes whether the parser collects the whole answer. It does not

establish that the parser handles non-text blocks correctly when the test never supplies any. A prudent implementation and evidence for that implementation are separate things.

The response also carries a stop reason and measured token usage. The initial path handles a completed text answer, represented here by `end_turn`; later tool use will change both what content can contain and what ending a response means. Treating an HTTP success as the whole semantic result would erase that distinction before the agent needs it.

No usable text is an error for this text-only conversation. Missing usage is also an error, but for a different reason: absent accounting is not a measurement of zero. Engine retains measured usage even when a response contains no usable answer. Agent can reject the exchange as conversation while Engine records that the attempt consumed tokens. History describes successful dialogue; accounting describes measured expenditure. Making them share one success flag would lose one of those truths.

Diagnostics travel through the owner chain. They omit credentials and remote error bodies, which can contain material that does not belong in local logs. An error needs enough context to identify the failed operation without turning the logger into a second transcript or a credential archive.

A long request is not necessarily a stuck request

Bill's account reports that a blanket HTTP client deadline killed valid model requests during later development. Generation can take a long time, including before response headers arrive. A

total-response timeout chosen for an ordinary web request can therefore become an accidental limit on the model's work.

This design retains connection and TLS protections but lets the owning context cancel the model request rather than imposing an arbitrary total-response deadline. The distinction does not ban deadlines. A caller with a real latency budget can express it through that context. It prevents a transport default from silently becoming the product's policy about how long thought is allowed to take.

Waiting for progress, deciding that a user should hear an update, and cancelling the request are different actions. Later runtime features will need to preserve that distinction. The first synchronous loop already benefits from getting the lifetime right.

It also makes exactly one API call for each valid question. There is no warm-up call, hidden summary, retry, or backoff. A rate-limit response is reported as failure rather than concealed behind another attempt. That narrow starting design leaves retry policy for a more capable client. A retry introduces policy about cost, delay, cancellation, and what may already have happened remotely. Leaving it out keeps the first loop's behavior inspectable.

The history has a running cost

Every successful turn extends the history sent on the next request. With a fixed system prompt and no pruning, the amount of retained conversation in the request grows even when the next question is short. For similarly sized turns, the cumulative amount of conversation sent grows roughly quadratically with the number of turns: early messages are paid for repeatedly.

Actual billing depends on the provider's tokenization, caching, and pricing. Request growth alone does not establish a dollar figure, nor does a fake server's character counter establish real token counts. The mechanism is nevertheless enough to explain why accounting belongs in the first implementation rather than in a future billing feature.

Engine sums the input and output usage reported by each response. Totals come from measured values rather than estimates derived from text length; missing data cannot be made reassuring by printing zero. Input and output remain separate because they represent different work and may have different prices. Even these totals are not omniscient: if an attempt fails without returning usage, the client has no measurement for it. A local counter is not a reconciliation with the provider's invoice.

Raw API access also comes with provider throughput limits and a pricing model that may differ from a packaged coding product. Owning the bytes buys control, not automatically cheaper operation. The design keeps the base URL configurable, alongside the key and model, so the same transport can address a compatible test server, a proxy, or the provider endpoint. Configuration changes the destination; it does not prove that every destination has identical behavior or trust properties.

Two front ends, one conversation

The CLI has two ways to use the same Agent. A human-facing chat mode supplies prompts and readable replies. A machine-facing mode accepts one JSON object per physical input line and emits one JSON answer per successful round, followed by cumulative usage at normal end-of-input. Neither needs its own conversation implementation.

For the machine interface, stdout belongs exclusively to the protocol. Logs and diagnostics go to stderr. A friendly greeting or a debug print on stdout is not harmless decoration; it is an extra message in somebody else's language. The initial CLI consequently makes machine mode the default and chat an explicit choice. That convention serves its callers; the more durable boundary is that presentation never leaks into the data channel.

Input handling has two unglamorous consequences worth retaining. Go's default `bufio.Scanner` imposes a token-size limit that can reject a valid long JSON question. A line reader avoids that accidental restriction. It can also return data together with EOF: a final line without a newline is still a line to process, exactly once. EOF with no remaining data ends the conversation. Other read errors end in failure, with a diagnostic and a nonzero exit, not the terminal usage record that denotes normal completion. Otherwise a damaged input stream can look like a successful session.

The source's live-chat account includes a particularly economical false success. A launcher consumed stdin; chat immediately encountered EOF and exited cleanly without making a request. The program had demonstrated that it could stop. It had not demonstrated that it could converse.

A real chat probes the whole route through the front end, Agent, Engine, and service. Controlled requests answer different questions: whether the exact prior messages survive unchanged, whether an assistant-introduced fact remains in later history, whether every returned text block reaches the caller, and whether usage is summed rather than invented. Neither a cheerful terminal exchange nor a green protocol check substitutes for the other.

The model's own account of its environment is another limited observation. In the source's preserved conversation, it could describe the raw-HTTP setup supplied in its system prompt but could not confirm how many times the program had sent the conversation. That restraint was appropriate. The prompt told it what kind of assistant it was; the program had not exposed its request counter. Inspecting outgoing requests establishes what the client sends. Asking the model what the client sends establishes what the model can infer from what it has been told.

The result is still a small chat client. Its conversation representation will soon be too simple. The useful inheritance is the separation underneath it: Agent decides what becomes history, Engine owns transport and measured usage, Ensemble owns application services, and front ends present the same conversation through different interfaces. Chapter 2 can replace the simple record without rediscovering who is responsible for it.

Chapter 2: One Log, Three Vendors

The second vendor was supposed to be the easy one.

In the CodeRhapsody history preserved by the original chapter, Bill began with a single-client agent. An interface was extracted from the Claude client, taking its existing methods as the requirements. Even the message types in its signatures belonged to Claude. When Gemini arrived, its implementation had to pretend to be Claude before it could be itself. The interface had concealed a client without supplying a common model of conversation.

The next implementation was a copy of the first, edited until it worked. OpenAI got another copy. The source account says each passed its tests when it landed; later, fixes had to visit three increasingly different homes and did not always reach them. The eventual redesign used a common context with separate vendor renderers and parsers, and the account reports substantially less code. That is useful evidence of a design problem, not proof that smaller code is always better.

Delivery supplied the qualification. The redesign shipped as a big-bang rewrite. At the point recorded in the source, capabilities still existed only in the old clients, and an audit found audio attachments broken across the new vendor paths. A cleaner seam had not completed the migration. The technical lead inherits both the better design and the obligation to account for what the replacement still cannot do.

An interface can hide a dependency without removing its assumptions.

Chapter 1 left a deliberate weakness in place. One owner kept a simple history, and one exchange established a reliable path from human input to assistant output. That settled who could change the conversation. It did not settle what a conversation was. A slice of role-and-text messages still lets the first API supply the answer.

This chapter replaces that answer without changing the established chat and stdio behavior. The durable record describes what happened. A context describes what those facts currently mean for the conversation. A renderer describes how one vendor needs to hear it. None is a synonym for the others.

That distinction matters more when an AI coding assistant can produce another client quickly. The mechanical work gets cheaper; the duplicated judgment gets harder to notice. The technical lead's review needs to follow ownership and information loss, not admire how quickly the new package compiles.

Three Things Wearing One Name

"History" is often asked to do three jobs at once: preserve the past, hold the current state, and serve as the next request body. Those jobs conflict.

A durable log should remember a large tool result even after the result has become useless to the model. The current context should be able to replace it with a small stub. A vendor request may then need to move that stub into a particular role or combine it with another entry. If all three are the same mutable list, a formatting fix can rewrite the past, and compaction can erase the evidence needed to debug compaction.

The source's remedy is an append-only event log, a reducer, and a vendor-neutral context. History owns the log and its reduction. Agent coordinates changes to it. Engine owns the live exchange and its measured usage, returning facts through Agent rather than editing History itself. Vendor code gets the two edges: rendering requests and parsing responses. It does not get a private copy of the agent's turn policy.

An ordinary turn makes those boundaries concrete. A human message becomes a recorded event. The reducer adds its content and moves the turn from idle to input-pending. Rendering uses that state and the relevant configuration to build a request. Recording the send moves the turn into flight. The response parser returns neutral events; those events enter the same log and reducer as the human message. A response without tool calls finishes the turn. A response with calls leaves tools pending, and the last returned result makes input ready for the next exchange.

The model's reply and the tool's execution are separate facts. A reply can *ask* for a tool without the tool ever running. The reply therefore preserves text and tool calls together, in the order the model produced them. A later `ToolCalled` event records dispatch; it does not duplicate dialogue. A tool return records the outcome. Losing that separation makes interruption history say more than the system knows.

Chapter 2's design can represent and replay tool events before it can execute tools. That is preparation, not a reason to install a job scheduler early. The data shape must leave room for the next capability; the runtime need not pretend that capability has arrived.

The log orders events by monotonic sequence number, not wall-clock time. Timestamps remain useful evidence, but clocks can disagree or tie. Ordering must survive both. The source uses readable JSON lines so ordinary tools can inspect the record

without knowing the application's types. A damaged conversation is an unfortunate time to discover that its evidence requires the damaged application to open it.

Recorded events are never edited. A redaction is another event. So is a failure. The recorded request event identifies the destination; it need not duplicate the derived request body. Reconstructing that body depends on the relevant state and rendering configuration. Today's configuration cannot be assumed to recreate every historical byte.

This also changes how a reviewer asks what is happening *now*. The original chapter records a test-harness bug that classified a request as mid-tool-loop whenever it contained any tool results. Full history was being resent. Once a conversation had used a tool, even later new turns still contained those results. The check mistook the past for the present. Current activity belongs to the state machine or the relevant end of the request, not a search over everything the model has ever seen.

A Reducer, Not a Second Opinion

The reducer's rule is small: the current context and the next event contain everything required to compute the next context. No call to the network, no secret inspection of Engine, no consultation with a second history owner.

This does not require copying every value. In Go, copying a struct containing slices can leave two apparent contexts sharing the same backing array. A renderer that edits one may quietly edit the other. A clearly owned, in-place reducer is safer than decorative immutability that is not actually immutable.

The turn states are likewise modest: idle, input-pending, in-flight, and tools-pending. A known event that does not cause a transition in the current state leaves that state alone. That makes

the reducer total without turning routine replay into a panic lottery. An *unknown event type* is different. The loader refuses it before reduction, rather than silently skipping a fact whose effect it cannot understand.

These rules put classification where the necessary knowledge lives. An input receiver knows that bytes arrived. It may not know whether the agent is already working, or whether those bytes belong to ordinary dialogue or temporary context. Letting the receiver decide produces several competing versions of turn state.

The source's temporary-context example is especially revealing. An ephemeral contribution arrives as recorded System input, but the reducer puts it in a pending slot rather than dialogue. Arrival alone does not start a model exchange. Another contribution replaces the pending one; an empty update clears it. Every arrival remains in the log, so replay can reconstruct the pending state without a second, unlogged input path. Neither replacement nor clearing changes the stable primary instructions.

The pending contribution is meant for one request. Rendering must not consume it: rendering the same snapshot twice should produce the same result. The recorded send consumes it in the reducer, including on a request that later fails. Otherwise an old hint can ride along on an unrelated turn. If the model quotes the hint in its answer, that answer becomes ordinary dialogue; seeing the words again does not prove the instruction was delivered twice.

The ordering matters. The request must be built from the state that still contains the pending contribution; then the send event advances the recorded state and the already-built request is used. Re-rendering from the consumed state would omit the very material the send was supposed to deliver. A failed HTTP exchange does not retroactively mean that no request was attempted.

This is a one-use *context* policy, not a claim of exactly-once network delivery. A log does not magically make an uncertain remote side effect certain. Its value is that the policy is explicit, replayable, and owned in one place.

Infrastructure errors and tool failures also take different paths. A connection reset or HTTP rejection is an exchange failure. A compiler that ran and reported a type error produced a legitimate tool result, marked unsuccessful. Treating that compiler result as a transport error encourages the agent to retry the same broken build as though the network might become more agreeable.

The parser cannot bypass this arrangement by updating context directly. It returns events. Otherwise it becomes a second reducer, with a second opinion about every invariant and no reliable reason to agree with the first.

Content Is More Than a String

A vendor-neutral conversation is not a vendor's message type with the names filed off. It preserves distinctions the agent needs, then lets the wire format compromise them locally.

The context has ordered entries attributed to human, agent, system, or tool. Each entry contains typed parts: text, tool calls, tool results, references to external bytes, opaque replay material, and redaction stubs. A single string cannot preserve those distinctions. Neither can a role borrowed from an API that calls a tool result a human message.

A tool call retains its issued correlation ID, its name and arguments, and the provenance of its producer. A result refers back to the call and keeps its error flag. Correlation is not cosmetic. The renderer cannot answer a call without identifying which call it is answering. Existing IDs survive even when their

spelling looks foreign to the target vendor. Only a genuinely absent ID needs a synthesized one, and that synthesis must be deterministic from stable event identity rather than random on every render. Multiple calls still need distinct, consistently matched identities.

Arguments must survive normalization with their meaning intact. The original account describes a JSON-number regression: routing a large integer through floating point rounded it. The resulting request remained perfectly valid JSON and asked for the wrong thing. Preserving JSON number representations fixed the problem; a targeted regression test detected the rounding when that protection was removed. Canonicalization is housekeeping, not a license to alter the model's request.

A blob part holds a reference, not the bytes themselves. Its kind distinguishes a local path, a remote URI, or a framework handle; its locator preserves the address. Base64 is a possible rendering choice, not a new truth to write into history. This chapter's renderer can refuse an unredacted blob it cannot represent. It need not fetch a URI or open a local file merely because a record mentions one. Representing a reference is not authority to dereference it.

The distinction survives redaction. A stub can replace the expensive content while retaining the original reference kind and locator. The next request sees less; the durable log still contains the original fact; later code still knows where the bytes live. "Removed from context" does not mean "deleted from storage," and is not a privacy-erasure guarantee.

The system prompt has a different status: it is generated output, derived from context and configuration, not a stored field inside context. A constant is an adequate generator at this stage. The ownership rule matters more than the sophistication. Hand-maintained tool descriptions tend to drift from actual tools; a mixed pile of generated and sacred handwritten text eventually

becomes impossible to change with confidence. Keeping prompt generation at the rendering boundary leaves room for later capabilities without making yesterday's prompt prose part of the permanent conversation model.

The source tests that distinction by adding renderers in a deliberately inconvenient order: Anthropic, OpenAI, then Gemini. The most alien shape comes last so that success cannot merely mean the last vendor was easy. The measure is changes forced into context, not implementation hours. A difficult renderer can leave the underlying model intact; a short renderer can conceal a missing fact.

The reported first-edition result was not a clean victory. OpenAI required no new context fields. Gemini required one. A function response needed its function's name, but that was already available on the earlier call with the matching ID. The renderer could look it up; storing the name again would have served only the wire. A call's thought signature was different. The context could hold opaque material belonging to a turn, but could not bind it to one particular call. That missing relationship required a field on the call itself.

The later implementation described in the source began with that lesson already incorporated and needed no further context fields. That shows successful reuse, not an independent rediscovery that the third vendor costs nothing. The durable review question is whether a proposed field preserves a fact that would otherwise be lost or merely duplicates one vendor's formatting. The signature relationship passes that test. The repeated function name does not.

Compaction Should Know What It Is Throwing Away

The original chapter describes Bill watching a Gemini SDK's built-in compaction summarize away the oldest portion of a session. The task was in the first message. After compaction, the agent remained fluent and confident but could no longer say what it was doing. Bill's workaround was a handoff document written by the agent for its successor. It became an ancestor of a later mechanism in the book.

The lesson is not that summaries never work. It is that age is a poor proxy for disposability. The oldest message may contain the reason every later message exists.

The source also reports measurements from CodeRhapsody coding sessions in which tool output and tool-call arguments dominated conversation volume. Those measurements motivate a category-based policy; they do not establish the composition of every agent's workload. A technical lead still has to measure the actual workload and identify which information carries continuity.

The distinction suggests an escalating set of redactions. First, replace tool-result content with stubs while retaining the calls. The conversation still records what the agent asked for, without repeatedly sending the entire answer. A stronger level removes calls and results while retaining the surrounding prose. Another can remove prose and reasoning. Finally, a summary replaces a span with compressed content.

These are different losses. A category purge can name what disappeared. A summary can lose a qualification or a task boundary in a way that becomes visible only when the agent makes its next decision. Tool results can contain indispensable evidence too; the category is a basis for policy, not permission to

discard indiscriminately. The shape supports these operations while leaving thresholds and selection to later context engineering.

There is a surprisingly consequential implementation difference between the first operations and the last. The non-summary redactions filter each entry in a span independently. A summary folds the whole span into one entry. The source's reference implementation initially treated all levels as cases inside the same per-entry loop. It copied the summary into every entry it was meant to replace. Compaction made the context larger. The machinery was tidy; the operation was wrong.

The corrected summary occupies one entry at the beginning sequence of the replaced span, attributed to System rather than impersonating one of the several speakers it summarizes. Its replacement text is recorded in the redaction event because model-written prose cannot be deterministically regenerated. Ordinary stubs can be derived from the superseded content. Replay therefore reproduces both kinds of compacted state without overwriting the source facts.

This is also a memory-ownership problem. A growing map of every sequence ever redacted looks handy, but duplicates facts already preserved in the log and has no natural bound. The current context needs the *result* of redaction, not a second lifetime registry of redactions. Dialogue itself can grow, but it has an intended bounding policy: compaction. An auxiliary map without such a policy is a slow leak waiting for a sufficiently long-lived agent.

Append-only history does not mean all storage is bounded. It means durable evidence and the working set have different lifetimes. Confusing those lifetimes is how a useful cache becomes a second database nobody meant to own.

Translation Is Allowed to Be Strange

One tool returned “ok.” Who said it?

In the API behavior documented by the source chapter, Anthropic represents the result inside a user message. OpenAI Chat Completions gives it a tool role. Gemini represents it as a function-response part in a user turn. The neutral answer remains: the tool said it. The wire gets the answer it can express.

This is where a renderer earns its keep. It is not merely a serializer. It reconciles a particular API’s constraints without exporting those constraints into the conversation model.

Anthropic’s ordering rule supplies the strongest example. A tool result must immediately follow the assistant message that requested it, and result blocks must precede ordinary text in the user message carrying them. The source distinguishes this from the tempting but inaccurate explanation that consecutive user messages are simply forbidden. The causal rule is adjacency and block order, not a generic demand for alternation.

The single-tool case makes a naive implementation look correct. The assistant requests two tools; result one arrives; a human adds an instruction; result two arrives. Appending rendered blocks in arrival order produces result, text, result. That is wrong for this wire. The renderer must finish grouping the results before placing the remaining content, preserving order within each group. History and context retain the actual arrival order. Only the disposable request receives the rearrangement.

The original account reports that an independent offline replay exposed this defect after the first tests had passed. That is a better review example than a vague request for “more coverage.” The reviewer changed the arrangement of valid facts until a hidden assumption became observable.

Missing results are a harder case. After an interruption, history may contain a requested call with no recorded return. That does not establish that the tool failed, succeeded, or even ran. A renderer must first look for an actual correlated result, including one recorded later. It must not rerun the tool to make the transcript look complete.

The bridge's chosen recovery policy is a deterministic, wire-only error result with the exact call ID and the text "No result was recorded; execution outcome unknown." The error flag describes the missing record, not a diagnosed execution failure. The placeholder sits with the other results immediately after the assistant's calls, ahead of intervening human text. A later real result replaces it in a fresh rendering. Neither the placeholder nor the reordered blocks become new historical facts.

This policy is a design choice, not evidence that an interrupted tool behaved one way or another. Pairing, adjacency, and block order must be validated before sending. If the history cannot be represented safely, a local error is better than malformed HTTP or an invented success.

Gemini has a different translation trap. A function response needs an object, not a bare string. Wrapping a text result solves only half the problem. If an already-object-valued payload is passed through untouched, its neutral error flag can disappear. An envelope has to preserve both payload and failure status on either path. "It was already JSON" is not a semantic guarantee.

Response parsing deserves at least as much scrutiny as rendering. Gemini's recorded stop signal can say `STOP` even when the response contains a function call. The reducer must inspect the parts to discover that tools are pending, rather than trusting a stop label that means something different on another wire. Equivalent responses should produce equivalent neutral

content and accounting while retaining their different provenance. Making the resulting records identical in *every* field would erase who produced them.

A smaller source failure illustrates the same obligation. OpenAI returned an assistant reply with an empty text string after reaching a token limit. The parser treated empty text as not worth recording. On the next request, the renderer emitted `null` where the empty string had been. That bare assistant message was rejected. The fake service had accepted it, so ordinary tests had concealed the loss.

Empty and absent are different facts. A seam that preserves tool calls but discards an empty text part is still lossy. Unsupported media must likewise cause an explicit error rather than vanish. Silent omission makes the output look reasonable precisely when a reviewer most needs it to look broken.

This ownership is why the design declines server-side conversation continuations. They can save retransmission, but the coding agent needs control over what history contributes to its next request. Redaction and compaction operate on that projection. The choice is a product tradeoff, not a claim that hosted conversation state is useless everywhere: this agent pays the replay cost to retain editing control.

Provenance Is Not Decorative Metadata

An opaque reasoning signature is not ordinary dialogue. It is material issued by a particular producer for replay under particular conditions. The context carries it without treating it as instruction or reconstructed prose. Readable reasoning text can remain in the log; the replay path preserves the opaque material and its origin.

That origin needs vendor, exact model, and API surface. A vendor name alone is too coarse. A routing alias requested by the client may also differ from the actual model identity returned by the provider. The client captures the actual producer when supplied, keeps the requested routing name separate, and uses an explicitly resolved replay target rather than guessing identity from a friendly alias. A target that has not been resolved is not permission to send a signature anywhere convenient.

This information has to be captured when the response arrives. A renderer looking back several turns cannot reconstruct a missing producer identity from the current model setting. A prewritten signature fixture can verify replay selection while saying nothing about whether the real parser captured the signature correctly.

The source records a useful surprise from its dated vendor probes. The initial story was that Gemini would reject another model's signature while Anthropic would quietly discard it. The reported tests did not support that simple division. Cross-model Gemini replays were accepted at the HTTP level; corrupted signatures and certain missing call-bound signatures were rejected. Anthropic's behavior depended on whether a receiving model could read the originating model's block, with some incompatible material quietly dropped.

Acceptance is not proof that a backend used the foreign material. The original chapter explicitly leaves that unobservable. These are observations from the source's September 2026 probes, not a current compatibility promise or fresh measurements by this companion.

The design consequently chooses the more conservative contract: opaque replay material goes only to the exact issuing model through the appropriate vendor surface. That policy is

stronger than “the server happened to accept it.” It avoids making agent continuity depend on a silent interpretation the application cannot inspect.

Call-bound material also stays attached to its own call. One call’s missing signature cannot be repaired by copying a sibling’s signature, and mismatched material is not made safe by a vendor-documented escape hatch. If the target requires material the stored call does not have, the renderer fails locally. A rendered request should not bluff its way past an invariant.

The source’s type choices support the same discipline. Vendor and surface are closed sets where the code needs exhaustive handling; model names are an open set compared for identity, not an ever-growing switch statement. An invalid zero value catches missing provenance rather than silently selecting a vendor. Readable serialized names retain the log’s inspectability, and unknown names fail on load.

Even “opaque” requires active restraint. The original chapter recounts a replay block whose decoded text asked for praise of the course author. The package carried it back to its issuer without obeying it. The anecdote is funny because the boundary is so small: bytes being transported do not gain authority merely because they turn out to contain readable words.

The Receipt Has to Add Up

Token accounting is another translation boundary disguised as bookkeeping.

The canonical usage record separates fresh input, cache writes, cache reads, and output. Those categories are meant to be disjoint. They have different economic effects, and stable-prefix design cannot be evaluated honestly if cached work is counted as fresh input or omitted altogether.

The source describes three different conventions. Anthropic's input count already excludes its separately reported cache categories. OpenAI's prompt total includes cached input, so cache reads and any reported cache writes must be subtracted to obtain fresh input. Gemini's prompt count likewise includes cached content, while its thinking tokens are reported separately from candidate output and must be included in output accounting. One response object can use a subset convention on the input side and a disjoint convention on the output side. A parser cannot infer arithmetic from the reassuring word "total."

The source's probes and API examples support those formulas in the chapter's stated context. They do not make prices permanent or eliminate the need to check a new surface's usage contract. The durable log records counts, not dollars. Rates belong in configuration so old sessions can be re-costed without rewriting history. Token counts alone also cannot represent time-based cache-storage charges; the source leaves that extension to later caching work rather than pretending the four counters cover every bill.

Malformed usage needs narrower failure than "throw away the response." Suppose the prompt total is known, the supplied cache-read field is invalid, and the output count is valid. Fresh input depends on the invalid field, so it is unknown too. The independently measured output remains evidence and must survive alongside a diagnostic about incomplete accounting. An omitted optional cache-write bucket can mean zero; a supplied value that cannot be parsed does not. Overflow, invalid signs, and broken subset relationships cannot be laundered into plausible totals either.

A partly broken measurement is neither wholly trustworthy nor wholly worthless. The sound accounting policy preserves what is independently known, identifies what depends on the

broken value, and keeps the uncertainty visible. A confident wrong cost report can influence decisions for months without ever crashing the program.

Historical usage in a replayed context is also not usage incurred by the currently running Engine. History derives the former from response events; Engine owns measurements of its own live exchanges. Offline rendering an old session must not pretend to have sent the old requests again. The distinction keeps accounting tied to events rather than to whichever object happens to be displaying them.

A Seam Worth Trusting

The architectural test is not whether three happy-path requests succeed. It is whether changing vendors forces the agent to change its account of what happened.

A useful review follows a fact all the way around the boundary: provider bytes become events; events reduce to context; context renders for the next exchange. The empty-string failure lives across that round trip. The large-integer failure lives inside normalization. The interleaved-results failure needs more than one call and an inconveniently timed human message. Each asks a concrete question that a pleasant demo does not.

Determinism supplies leverage. Replaying the same log through the same reducer should produce the same context. Rendering the same context with the same configuration should produce the same bytes and leave the input untouched. Random IDs, unstable map iteration, injected current timestamps, and accidental mutation all undermine that property. Ordered parts are semantic order; sorting them for prettier output would merely make the wrong result reproducible.

A valid replay record also needs a format version and strict interpretation. Current code replays the log; it does not resurrect historical binaries. The version belongs in a header rather than a sequenced event. The source allows a missing header to mean the current format, but refuses facts it cannot interpret: unknown event kinds, parts, actors, or vendors. Sequence numbers must be increasing. A legacy blob's bare path is refused rather than guessed into a reference kind, and an unspecified kind is invalid. A local-path guess might appear to work until a URI becomes a filename that never existed.

Starting with a broken log and silently skipping the awkward line creates a context no honest sequence of events produced. Redaction changes the projection through its own appended event; it does not rewrite earlier events. Wire-only recovery changes no events at all. A later dump retains the original facts. The log is the evidence, not a workspace for the renderer.

Offline rendering uses the same target configuration as live sending but needs neither a credential nor a reachable endpoint. Credentials belong in neither the durable log nor the rendered request body. That separation makes a saved conversation useful for inspecting a proposed request without accidentally sending it. Saving events still does not establish automatic session resume or crash-safe journaling; those are additional behaviors, not properties bestowed by a JSON-lines file.

The source records a particularly revealing test gap: removing the reference implementation's only use of call-bound replay material initially left all checks passing. The fixtures tested opaque material belonging to a turn, not to a call. The distinction that had forced the new field was the distinction the tests never exercised. A foreign-origin fixture could correctly withhold a signature and still prove nothing about replay to its issuer.

A later review found the next gap: a saved-signature fixture could pass while the real parser discarded new signatures. Capturing material, associating it with the correct call, withholding it from other targets, and replaying it to its issuer are separate obligations. Likewise, a secret-free rendered redaction stub cannot prove that the reducer preserved the original locator internally. The observation has to reach the property being claimed.

Offline fixtures are valuable precisely because these cases are awkward to provoke live. They cannot establish that a provider actually honored replay material or that a fake has enforced every vendor constraint. Live observations answer different questions, with their own limits. The strongest review does not confuse either form of evidence with the other.

The payoff is not universal compatibility. New APIs will have new limitations, and some histories cannot safely cross a particular wire. The payoff is a place to put that difficulty without spreading it through the agent.

The durable log remembers what happened. The reducer decides what those facts mean now. The renderer translates that state for a particular listener and refuses what it cannot represent honestly. Once those jobs are separate, the next capability can arrive without asking a vendor’s message struct for permission.

Chapter 3: Six Tools, Ninety-Two Percent of an AI Coding Agent

Bill counted the tools after months of using them.

The CodeRhapsody corpus described in the original chapter contained 511 sessions and 70,401 tool calls. Five tools accounted for roughly 91 percent: running commands, reading files, editing files, searching, and writing. Adding directory listings brought the share to 92 percent. Fifty-one available tools, and most of the work fit in six.

That is evidence about Bill's recorded workload, not a minimum viable agent established by nature. The logs were private, and several newer tools, including much of the sub-agent suite, postdated the measurement. Their absence could not establish their uselessness. Even the directory-listing tool earned its place partly through judgment: orientation is cheap; guessing paths is not.

The more revealing result was what happened *within* those ordinary operations. Targeted edits outnumbered whole-file writes about seven to one. Yet the frequently used edit tool had a defect Bill discovered while writing the chapter: when an anchor appeared more than once, it changed the first occurrence and reported success. The zero-match case refused correctly. The many-match case guessed quietly.

Chapter 2 gave the agent a durable account of conversation, a reducer, and vendor-specific request boundaries. It could preserve a request to run a tool without running anything. This chapter connects that recorded intention to effects: files read,

bytes changed, commands executed. The difficult part is less the filesystem API than making “requested,” “ran,” “failed,” and “finished” mean different things all the way around the loop.

A Tool Call Is the Middle of a Turn

Consider an ordinary repair. The human asks about a failing test. The model explains that it will inspect the failure and requests a command. The agent records the response, runs the command, records its output, and sends the updated context back. The model may then read a file, edit it, and run the test again. Only when a response contains no more tool calls does the turn finish.

The human has supplied one request. The model and tools may have exchanged several rounds underneath it. Returning after the first tool result would stop the repair just as the model acquired the information needed to continue.

The conversation remains Chapter 2’s conversation. `ResponseEnded.Parts` preserves ordinary text and calls together, in their original order. A call part records what the model asked for; dispatch and return events record what the runtime did about it. Agent coordinates those facts through History. Engine continues to own the vendor exchange rather than acquiring a second history or a private tool-loop policy for each provider.

The caller receives only the final response’s text. Intermediate narration still belongs in history, but it is not part of the final answer returned through the existing interface. The source’s student implementation initially concatenated “I’ll check” with the eventual answer. That was not lost data; it was data returned through the wrong interface. Preserving the conversation and selecting the answer are separate responsibilities.

Every result refers to its call's ID, not merely its name or its position. Two reads of the same file can ask for different ranges. Two calls in one response can fail differently. When a provider omits an ID, the parser assigns a deterministic, distinct identity while capturing the call, using stable event identity and part position. Waiting until rendering is too late: dispatch already needs the identity that the log and eventual result will share.

Calls execute one at a time, in the order requested. This is an intentional constraint, not an unfinished concurrency feature. A command can change the tree that the next read observes. Running the same calls concurrently changes the meaning of the response unless their independence has been established elsewhere.

The loop also has a round limit. The source uses sixteen rounds; the important property is a finite bound on repeated model exchanges, not that particular number. A model can keep asking, and a scripted fake can keep repeating its last call. The bound prevents an endless conversation with the tools. It does not interrupt a command, impose a deadline, or make a blocked shell responsive. A single round can still last forever.

One Registry, Two Views

Before a model can request a tool, the request must declare it. A declaration supplies a name, a description, and an argument schema. The implementation adds the function that performs the operation. Those are two views of the same registered tool, not two catalogs maintained by different people.

The registry belongs to an Agent. The composition root installs its definitions; both rendering and dispatch consult that same set. Otherwise a model can be offered a tool that cannot

run, or dispatch can expose a capability the Agent was not meant to advertise. A mutable global registry would make one Agent's configuration someone else's surprise.

Six tools need little machinery. An ordered collection with declaration and dispatch operations is enough. The source places the shared definitions in `internal/common` and the implementations in `internal/tools`. A handler receives its actual owner through a narrow `ToolContext`, which exposes Engine and the existing ownership chain. That gives a diagnostic access to the services already owned by its Agent without globals, copied sibling services, or a fictional parent object. A synchronous function call does not need a new tool runtime to feel architectural.

An ordinary library client can still construct an Agent with no tools. In that case the renderer omits the declaration field, preserving conversation-only behavior. The command-line application installs the six tools, but no new command or alternate transcript convention is required. The existing chat, stdio, dump, offline-rendering, and log configuration remain the public interface. Capability grows beneath it.

Declaration syntax belongs behind the same vendor seam as messages. The shared schema describes the local argument contract; Engine projects it into the provider's accepted dialect. The Edition 3 live exercise exposed a concrete difference: Gemini's `parameters` rejected top-level `additionalProperties`. The repair omitted that keyword from the Gemini projection for these built-in object schemas. It did not loosen local argument validation, alter the other providers' declarations, or grow a general recursive JSON Schema translator.

That narrow fix matters. A provider's inability to express a constraint does not make the constraint disappear at execution time. Required fields still have to exist with the right types.

Empty `content` or `new_text` is a valid string; a missing field or `null` is not. A structurally valid API response can carry the wrong argument types. Local validation has work to do even when JSON parsing succeeded.

An empty registry does not excuse an unanswered call either. The source's student initially returned early when there were no tools. An unexpected request then remained in history with no result. The correct response is a correlated error explaining that the tool is unavailable. A declaration governs what should be requested; dispatch must still handle what actually arrives.

Failure Must Come Back

A missing file is a tool result. An unknown tool is a tool result. Arguments of the wrong type produce a tool result. Each carries the original call ID and an error indication, so the model can correct its next action.

Dropping the result on an error leaves a hole in the conversation. The model requested something, the runtime knows why it did not happen, and the next exchange receives none of that knowledge. Crashing the whole agent is often little better: a recoverable misunderstanding becomes a lost turn.

A command's nonzero exit status is different. The shell successfully ran the test suite; the test suite failed. That is ordinary command data, not a malformed tool invocation. Chapter 2's distinction between an exchange failure and an unsuccessful operation now needs this finer resolution: a failed test does not set the tool-error flag merely because the command returned nonzero. The model needs to repair the program, not its request to run the tests.

The blocking command tool therefore reports stdout, stderr, and the process exit code separately. Parsing an error-looking sentence out of stderr is not an exit-code implementation. The source's original check accidentally demonstrated why: `go run` of a program that exits 7 itself exits 1 and prints `exit status 7`. A check looking for the number in captured text passed even without an explicitly reported exit code. A silent shell `exit 7` exposed the missing observation. The wrapper's status and its child's status are different facts.

Results then travel through the normal history and rendering path. Anthropic's result-first ordering remains a renderer obligation, not a reason to reorder historical events. Its rule needs a request containing both tool results and other content to become observable. This synchronous chapter cannot naturally receive a new human message while blocked in a handler, so an ordinary session can pass without ever testing the awkward ordering. A targeted mixed-content fixture reaches the property that the happy path cannot.

The distinction also prevents accidental reliance on Chapter 2's recovery placeholder. A wire-only "outcome unknown" result is useful for replaying an interrupted record. It is not a substitute for recording a failure that the live dispatcher actually observed.

Explanation Before Effects

Some model policies require an explanation before tools. Stating that requirement in the prompt is useful; enforcing it immediately before dispatch gives it operational meaning.

For a selected model, the same response must contain nonblank ordinary assistant text before its first tool call, explaining what will run and why. One explanation can cover a coherent batch. An explanation from an earlier turn does not

describe this batch. Tool output, hidden thinking, and a readable reasoning summary are not substitutes. Text after the call is too late.

The runtime can check presence and order. It cannot establish from those checks that the explanation is adequate or that the proposed work is authorized. This is an explanation policy, not a safety proof.

If the batch lacks the required preceding text, the dispatcher refuses the whole batch before invoking any handler. It preserves the response and returns an error for every call, explicitly saying that nothing ran and requesting an explained retry. Refusing only after the first operation would turn a precondition into a commentary feature. The retry consumes the existing round budget, so a model that repeatedly omits its explanation cannot create a second, unbounded repair loop. Models outside the selected policy retain their ordinary behavior.

The Edition 3 Gemini work found a parser bug at exactly this boundary. The inherited parser treated signed ordinary text as hidden thinking. That produced blank answers and false explanation refusals: the model had supplied visible prose, but the application had hidden it.

A signature does not turn ordinary text into reasoning. The text stays visible and portable. Its signature stays bound to that exact text part and to its source vendor, API surface, and model, and is replayed only to that same source. Genuine hidden thinking remains hidden. Chapter 2's provenance rule therefore protects two different things here: private replay material from being sent to the wrong issuer, and ordinary language from being swallowed by a parser that classified it by the wrong clue.

The decisive observation for refusal is an effect that did *not* occur. A handler that would leave a visible change makes this test meaningful: the unexplained batch leaves nothing behind, and an explained retry executes. Counting returned errors alone cannot

establish that the runtime refused before acting. Streaming will later make arrival order another part of the problem; this chapter has complete responses to inspect.

File Tools Spend Attention as Well as Bytes

A shell can implement all six operations. It can read, search, list, replace, and write. Bill asked whether anything beyond `run_command` was necessary, and in the capability sense the source's answer was no.

Capability is not the most useful comparison. Dedicated tools change the cost of expressing an intention, the precision of the result, and what happens when the intention is ambiguous.

`read_file` accepts a one-based inclusive line range and an output-size cap. The range is selected before the cap is applied, and truncation is reported rather than hidden. The source's accepted implementation uses the whole file by default and a 32,768-byte cap; that value is a documented choice, not a universal optimum. A bounded read makes it possible to ask a narrow question without importing a large file into every subsequent request. Chapter 2's full-history exchange makes that cost cumulative until context management removes the material.

`search_files` makes those narrow reads discoverable. Its regular-expression matches use familiar grep notation: `path:line:text` for matching lines and `path-line-text` for surrounding context. Overlapping or adjacent context windows merge, and separate groups receive `--`. The output format is already legible to the model and to the human inspecting its work; a proprietary search dialect would buy little here.

Context defaults to zero lines. The historical corpus showed that the model frequently supplied an explicit context request, often asking for more than the then-default of two. That supports a cheap default for this workload, not a general claim that surrounding code is unnecessary. A model can request more. Extra text already admitted to history has already spent attention and tokens.

`list_directory` lists immediate entries rather than recursively filling the response. Sorted names and a trailing slash on directories make orientation stable and cheap. `write_file` can create parent directories and new files, but refuses replacement of an existing file unless `overwrite` is explicitly true. The refusal identifies the existing target and its size. Append preserves existing bytes and needs no overwrite flag; an empty content string remains a meaningful input.

That guard protects against one easy mistake: presenting replacement as though it were creation. It is not backup, and the explicit flag is not evidence that the model read or understood the old content. A shell can overwrite the same file without using the guard. The design makes the dedicated operation more specific; it does not make arbitrary shell access safe.

The original account gives less abstract reasons too. During work on the book, the authoring tool refused edits when required heading context was absent or a wrapped word no longer matched an anchor. The refusal created a chance to inspect instead of guessing. Shell editing also carried platform differences and quoting hazards, including backticks being interpreted while preparing text meant to be written literally. File tools remove those distractions from the model's task. They do not make the intended change correct, but they reduce the number of unrelated ways to express it incorrectly.

The seven-to-one preference for targeted edits is consistent with that argument. It shows what agents used when both operations were available in this corpus. It does not, by itself, isolate the tool's causal effect from every other feature of the workload. The useful design hypothesis is narrower: affordances shape behavior, so removing one because the shell is theoretically sufficient can change the work an agent actually performs.

An Anchor Must Identify a Place

The edit defect gives that hypothesis teeth. A tool can offer the right operation and still define success badly.

When an anchor fails to match, several policies are coherent. Strict refusal preserves the file and asks for a better anchor. Fuzzy matching can tolerate stale context but risks selecting the wrong region. Falling back to whole-file replacement can finish the change while greatly widening its scope. The choice affects recovery cost, predictability, and how much authority a short argument grants the implementation.

The same decision has to cover multiple matches. Refusing zero matches while silently choosing the first of several is not a strict policy. It is guesswork with a well-tested error message on one side.

Bill's original `edit_file` did precisely that. A single replacement with a limit of one made the common case look correct and the ambiguous case look equally correct. The source reports the defect as a discovery during authorship, not as a repair whose later deployment has been established here.

The second-edition student chose exact, unique anchors and found a smaller trap. In `aaa`, the anchor `aa` identifies two starting positions. A routine counting non-overlapping

occurrences reports one. It answers how many disjoint replacements fit, not whether the location is unique.

Under the student's policy, both zero and multiple matches leave the file unchanged. A zero-match refusal identifies the file and suggests rereading it. A multiple-match refusal gives the actual count, including overlapping sites, and a bounded list of one-based starting line numbers. If that list is truncated, the response says so while retaining the full count. The tool does not quietly relax exact comparison to be helpful.

These details turn refusal into useful feedback. "Could not edit" avoids damage but gives the model little basis for a better request. A count and locations expose the ambiguity without returning the entire file. The model can choose a more specific anchor rather than spend a round discovering what the tool already knows.

Strict matching is not the only defensible design. The source also describes CodeRhapsody's line-replacement tool searching near supplied line numbers. A localized repair can reasonably trade exactness for tolerance. The tradeoff should be explicit rather than smuggled into an error path.

Text anchors have a compositional advantage. An edit near the top of a file shifts later line numbers, while an anchor elsewhere can remain valid. In the reported corpus, anchor edits outnumbered line replacements roughly thirty to one. The mechanism explains why anchors can support several sequential changes without rereading every shifted location. The count records a preference, not proof that line-oriented editing is always inferior.

A Small Tool Set Is Not a Sandbox

Per-Agent registration creates a place to control which operations are exposed. An Agent with only reading, searching, and listing tools lacks the declared file-mutation and shell operations. That is useful capability separation.

It does not establish filesystem confinement. Read tools may still reach sensitive paths. A launch directory determines where relative paths begin; absolute paths and parent paths can reach elsewhere. A shell bundles many capabilities behind one tool name. Calling it `run_command` does not narrow the authority of the process it launches.

This chapter establishes no general permissions system. Dedicated tools make later restrictions expressible, but selecting a small registry or changing directories is not equivalent to enforcing those restrictions.

There is one narrow credential boundary now. Provider keys belong to the HTTP transport, not to the child shell. Before launching a command, the implementation removes the supported key variables `LLM_API_KEY`, `ANTHROPIC_API_KEY`, `OPENAI_API_KEY`, and `GEMINI_API_KEY` from the shell environment while retaining ordinary build settings. A fresh shell starts on each call, so a `cd` in one invocation also does not silently alter the next.

Environment filtering prevents this specific inheritance path. It does not remove credentials from files, other variable names, or resources the process can otherwise access. Dummy key values observed through the actual launched shell can establish that the named variables were removed without exposing a real secret. A helper tested in isolation cannot prove that the launch path used it.

The source's first book-repository demo supplied a modest warning about effects. The agent invented a project codename and saved it in the repository root. The demo worked and left a file behind. Nothing had to crash for the run to exceed its intended footprint. Disposable workspaces and recoverable source state limit the consequences of a demonstration; they do not become containment merely because they are convenient.

A Fake Can Agree Too Easily

A deterministic fake makes the loop inspectable. It can serve text plus multiple calls, missing IDs, recoverable failures, and further rounds before the final answer. Unlike a live model, it can repeat the same awkward arrangement every time. Its value lies in stating and exercising a protocol, not in responding intelligently to the prompt.

The source's fake was generous in a particularly damaging way: it volunteered tool calls whether or not the request declared tools. An implementation could pass the loop checks without sending declarations, then do nothing useful against the real Anthropic API. Bill found the omission by auditing the chapter. The test world had supplied a capability the implementation never requested.

Chapter 2 supplied the opposite generosity: the fake accepted an empty assistant message rendered with `null` content where the real endpoint rejected it. A fake can send too much or accept too much. Either way, agreement between implementation and fake does not establish agreement with the provider.

A fake does not automatically discover that the real system changed. Focused observations of the real API supply missing facts, and those facts must return to deterministic coverage. This

is an outer loop for resolving concrete uncertainty, not a reason to make every edit depend on a slow, metered, nondeterministic model call.

Even inherited fixtures can become wrong when the product gets better. Chapter 2's fake ended with a tool call because that chapter only recorded calls. Chapter 3 answers it and asks for another response. A fake that repeats its last reply then drives the new loop into its round limit and changes accumulated usage. Extending the script with a final, zero-usage reply preserves the earlier accounting assertions while completing the newly possible protocol. Removing the assertions would hide the change rather than explain it.

The useful review question is whether the observation can distinguish the promised behavior from a plausible broken version. A single tool call cannot expose miscorrelation between two calls. An existing-file refusal alone cannot prove new-file creation remains allowed. An error result cannot establish that a forbidden handler never ran. A canned signature cannot establish that the parser captures newly arriving signed text. Each distinction needs a situation in which getting it wrong becomes visible.

What a Live Repair Establishes

The source records the second-edition student driving its own binary in three disposable Go projects, through Anthropic, OpenAI, and Gemini. Each project contained a greeting returning `helo` and a test expecting `hello`. The models listed the directory, ran the failing test, read the source, made a targeted edit, and reran the test. Inspection found the intended change; the command results moved from exit code 1 to exit code 0.

Those observations support more than an assistant saying "fixed." The call records establish which operations were requested and returned; the inspected diff establishes the changed bytes; the second test result establishes that the tested behavior passed. A later recall response showed that conversation context survived the work. None of these observations substitutes for the others.

The error demonstration initially failed to demonstrate an error. One model interpreted "No more tool calls" as forbidding the requested read; another declined to read a nonexistent file. Clearer follow-ups produced actual missing-file results. A model declining to act and a tool acting unsuccessfully are different paths, even when both leave the filesystem alone.

An additional bounded session exercised search, write, and read-back, bringing all six tools into the recorded live work. The source explicitly says that the live build preceded the overlapping-anchor and empty-registry corrections. Those fixes had deterministic coverage; the earlier live run is not evidence that the later binary exercised them. This companion reports the source account, not a new reproduction of those runs.

The Edition 3 additions sharpen the diagnosis further. Tool declarations could fail at Gemini's schema boundary. Visible signed text could disappear in parsing. Separately, a missing resolved model identity could suppress private replay material even with a correct parser. The launch and harness omitted `LLM_RESOLVED_MODEL`; correcting that setup required no source change. The resolved identity comes from an observed returned producer on the same vendor, API surface, and route, not from assuming the requested routing alias is the issuer.

These failures can look similar from the final answer and require different repairs. A model listing establishes what a credential can see, not which model supports tools on the chosen API surface. A newer or larger model name settles neither

suitability nor cost. An unknown replay target correctly withholds private extras; it does not prove the parser lost them. Diagnosing from the boundary that actually failed is cheaper than rewriting a working loop.

The resulting agent can now close a coding cycle: inspect, change, execute, interpret, and continue. It remains deliberately synchronous. A long test leaves it waiting inside the handler; a server that never exits prevents the turn from returning at all. Sixteen allowed rounds offer no consolation during round one.

That is the next chapter's problem. The small job-control share in Bill's historical count understates its importance: the next design changes the behavior of `run_command`, the most-used tool in the corpus. The agent already has reach. What it lacks is a way to keep observing and responding while the work continues.

Chapter 4: Jobs, or Why a Tool Call Is a Process You Supervise

The screenshot tool was supposed to grab the framebuffer and return.

In the first-edition incident preserved in the source chapter, the narrator asked for a screenshot. The tool call never returned. The agent still looked busy, but its turn never completed and no observer fired. Bill had to kill the process. That killed every agent in the tree, including sub-agents doing unrelated work without trouble.

That it was a screenshot matters: this was neither a shell command nor a network fetch. A repair confined to shell execution would have missed the actual hang. Substitute a test suite, a development server, or a tool supplied by somebody else, and the underlying defect remains: the caller has handed over control until the callee chooses to return. Chapter 3's tool loop can survive a failed command. It cannot survive one that never finishes answering.

The tempting repair is a timeout. Give everything thirty seconds, then cancel. That certainly returns control. It also kills the healthy thirty-one-second build and makes the development server impossible by definition. A hung screenshot and a running server can both be silent; elapsed time alone does not tell you which one you have.

Bill's more useful question is not "How long may this operation live?" It is "How long should I wait before hearing from it again?"

Those are different clocks. Keeping them different is the chapter.

A Handle Instead of a Hostage

The design introduces a job between calling a tool and receiving its eventual result. The job owns an identity, accumulated output, lifecycle state, and enough machinery to observe or control the work. The first call may still return a completed result immediately. If the work takes longer, the call returns a running report and a handle instead. Work continues under supervision.

Three seconds is the chapter's default wake interval. It is a policy for returning control, not evidence that a process has misbehaved. A report can truthfully say that job 3 is running, has produced no new output, and has stored zero bytes so far. The model can wait again, do something else, send input if the tool supports it, or explicitly kill the job.

The useful change is that silence is now observable. It no longer confiscates the conversation.

This also avoids splitting the tool vocabulary into foreground and background variants. The model need not predict whether a command will finish quickly before launching it. A fast read looks much like the old read. A long command acquires a handle when the wait ends. The lifetime belongs to the job in either case; the difference is how much the caller observes during its first visit.

That separation must hold below the user interface. A context deadline attached to the work would recreate the timeout design, however politely the report said "still running." Likewise, cancelling a waiter must not quietly cancel the operation it was observing. Killing work is a distinct decision with its own recorded reason.

Jobs here do not survive application exit. They outlive an individual wait, not the process that owns them. Ensemble still has to shut them down. This is supervised work inside one application, not a durable distributed queue smuggled into a coding agent.

The Shell Is Not the Boundary

A shell command is the obvious place to notice a hang. It is the wrong place to put the whole remedy.

The source uses shell-only supervision as a negative control. Its two file-tool checks are load-bearing: remove either the `read_file` job-model check or the large-output check, and the wrong implementation can score 100. That is a measured weakness in what the reduced test suite can distinguish, not the original implementation's score. Without the common job path, a large file result can flood the model instead of receiving the bounded preview, and a blocked read can still freeze the loop.

The design belongs at dispatch, where *every ordinary tool* enters execution. Reading a file, listing a directory, launching a command, and invoking an application's custom tool all pass through the same job lifecycle. Otherwise the framework can only promise responsiveness for tools whose internals it happens to know. The screenshot might live behind a foreign tool. A registry entry should not be able to opt the whole agent back into captivity merely by having a different name.

This makes ordering important. A job's handle must exist before its handler can have effects. The `tool_called` event carries that handle before execution begins. If a tool writes a file and then fails, the record already identifies the job responsible.

Allocating a handle only when the first wait expires would leave fast calls, early failures, and effects before the timeout outside the supervised account.

In the chapter's model, Ensemble allocates integer handles so its Agents cannot accidentally reuse them. Each Agent owns its own jobs collection and output directory; a job retains that collection as its parent. The specimen `cr/io/3` belongs to the CLI's data-directory convention, not a global storage location for every embedded Agent. The handle names supervised work, not a process ID and not a promise that the underlying operation is a shell.

Admission and execution also have to agree during shutdown. If shutdown closes admission while a call is arriving, the application cannot accept new work after deciding that all outstanding work has been cleaned up. Nor can cancellation remove a job from view while its delayed startup quietly launches a process afterward. Cancellation before startup must prevent execution from beginning; otherwise the application has lost track of work it still allowed to start.

A goroutine is not a lifecycle. "We start it asynchronously" explains how a caller escapes blocking; it does not explain who owns the work, whether it may still start, or who will clean it up.

There is a small, necessary exception to the ordinary-job path. The four supervision tools—`wait_for_job`, `send_input`, `kill_job`, and `tool_limits`—operate the machinery directly. Waiting for job 1 must not allocate job 2 to wait for job 1. Killing a job must not create another job that now needs killing. Their calls remain part of conversation history, but their own call records do not acquire job handles.

This is a semantic bypass, not a second execution engine. Ordinary work enters the common lifecycle. Control operations inspect or alter that lifecycle without becoming recursively supervised work themselves.

Observation Has Policy Too

Once a call can return while work continues, “wait” needs a more precise meaning than “sleep for a bit.”

The source’s watchdog history shows why this policy belongs to the caller. An early repair tried to infer how long each tool might block from a name-to-timeout map and its description. A missing tool got the default; a healthy tool whose own wait exceeded that default could be abandoned mid-wait. The next repair made tools declare their blocking contracts. That repaired the missing information but kept the question that required the table.

Bill changed the question. “How long may this tool block?” demands knowledge about the tool. “How long before I want to look again?” is a choice on this call. A caller can look shortly after launch, wait longer for a build, then choose differently on its next observation. The missing-row defect has nowhere to live in that design. This is more valuable than maintaining the timeout map more conscientiously.

The chapter has three observation controls: a delay, an optional output pattern, and a maximum inline byte count. A call may ask to wake after a particular interval, when unseen output matches a regular expression, or when the job ends. The byte cap limits how much comes back in the report. None of these authorizes the work to run longer or grants it additional capabilities. They govern the caller’s view.

The precedence is deliberately short: defaults, then any pending `tool_limits`, then the call’s own arguments. The waiting tools expose observation arguments directly. Other local tools, and tools whose schemas the framework does not own, may have nowhere to put them. `tool_limits` supplies that missing route by staging limits for the very next call. It is not a persistent Agent or Ensemble policy setter.

“Next” means the next attempted tool call, not the next convenient target. A `kill_job` consumes the setting. Another `tool_limits` consumes it. An unknown tool or malformed call spends it too: dispatch applies and consumes pending limits before validation or execution. Calls consume them in model-response order, even when one response contains both setter and target; this is about dispatch order, not which job finishes first. An explicit argument wins for its call without becoming another pending setting.

A friendlier rule might save limits until the next job-creating or waiting call. It would also leave the model remembering which categories count and wondering whether a setting from several calls ago was still armed. The source rejects that hidden persistence. One setting, one next call, then gone.

Two consecutive setters expose a subtle consequence. The first setting governs observation of the second call and is spent. The second creates a fresh pending setting from defaults plus only its own explicit fields. If the first raised the byte cap and the second sets only a delay, the elevated cap does not carry forward. Otherwise replacement would quietly become accumulation, and an omitted field would retain a history the model could no longer see.

Literal rules can still misfire. The source records a live model calling out the danger: “it's easy to burn it on the wrong thing.” Set a generous cap for a file read, glance at a directory first, and the directory listing gets the allowance. The subsequent truncated read looks inexplicable unless the consumption is visible.

The repair is a diagnostic, not a target-matching language. Every consuming call’s result begins with a line naming the tool that consumed the limits and the delay, pattern, and byte cap applied. This includes the four control tools that are not jobs. The setter’s reply also warns that whichever call comes next will

report consumption. A mistaken burn now explains itself in the result that caused it, close enough to correct without reconstructing several rounds of hidden state.

The argument details preserve that local control. Fractional delays are useful for a quick look, and zero means look now; a negative delay, nonpositive byte cap, or invalid regex is an ordinary tool failure. A pattern selects the output worth waking for while the delay still bounds the observation. None of these choices should turn into an undocumented persistent setting.

The distinction from lifetime limits remains absolute. “Wait up to thirty seconds for this report” does not mean “permit thirty seconds of execution.” Changing the observation policy cannot accidentally become permission to kill.

A Wait Is a Read With Consequences

A job report is not a subscription to all future output, nor is it a fresh copy of the complete transcript each time. Each job has a cursor. A wait reads output after that cursor and advances it.

This is destructive observation of the incremental stream, even though the retained output file is not destroyed. Two callers waiting on the same job share consumption: one can drain bytes the other would otherwise receive. If a second observer needs the complete history, the saved file is the appropriate surface. A wait handle is not secretly a broadcast channel.

The distinction matters most with patterns. Suppose a debugger has already printed `(dlv)` and the caller has consumed that prompt. After sending the next command, a wait for `(dlv)` must match a *new* prompt. Matching the entire accumulated file would let the old prompt satisfy every future wait immediately. The debugger would appear miraculously responsive while the model raced ahead of it.

Patterns therefore apply to unseen bytes. They also change what counts as a wake. On a later `wait_for_job` without a pattern, unseen output must return control promptly rather than notify the waiter only for it to go back to sleep. With a pattern, arbitrary new bytes are not enough; the requested match is the interesting event. Completion still ends the wait even if the pattern never appears, and the delay bounds how long the caller waits when the job remains live. A missing prompt should not conceal the fact that the process has already exited.

These distinctions give the predicates their practical precedence: terminal work can be reported as terminal; a requested pattern replaces "any output" as the output condition; the timer returns control without changing the job's state. They are not three independent reasons to abandon whichever check happens to be inconvenient. In particular, a fragment arriving before a prompt should not defeat a pattern wait, and a timer firing should not turn a live process into a failed one.

Waiting on an already-finished job is valid. It may return retained unread output, or no new output if the cursor has caught up, together with the terminal state. Requiring a job to be live would turn a harmless race—completion just before the wait—into an API error. It would also make the natural sequence "launch, inspect something else, collect the result" unreliable whenever the work was fast.

The initial no-pattern observation deliberately differs from that later wait. It preserves the fast full-result path instead of returning at the first output fragment. When a job completes within the wake delay and under the cap, with no directory override or consumed-limit diagnostic to disclose, its first report preserves Chapter 3's bare result. The PTY's merged shell output is an explicit formatting exception, while exit-status information remains. This makes ordinary short work familiar without hiding

new facts. It is not permission to block indefinitely: when the wake delay ends, unfinished work yields a running report. Later observations can then return as new output arrives.

This is where the interface stops being just "async." The hard part is deciding what the observer has consumed, what event it is waiting for, and which facts remain true after the observation ends.

Keep the Bytes; Spend the Context Carefully

Output storage and output presentation have different budgets.

The chapter keeps a job's full output under a locator such as `cr/io/3`. The inline report has a default cap of 16 KiB, adjustable per call. When output exceeds the cap, the report gives a bounded preview, the locator, and accurate byte accounting. A model can decide whether to read more instead of receiving a megabyte of compiler complaints as an unsolicited contribution to its context window.

The cap is not a retention limit. Truncating the saved file would destroy precisely the evidence needed to investigate an abbreviated report. Conversely, keeping the full file does not justify sending all of it to the model. Disk and conversational context serve different readers at different costs.

The cursor and cap must not be confused either. A report consumes an interval of new output; its preview may omit some of that interval. The omitted bytes remain available through the file. A caller should not assume repeated waits will eventually reproduce every omitted byte inline. That would make a presentation cap behave like an undocumented pagination protocol.

The source's flood fixture is useful because it distinguishes these surfaces: the complete retained output must include its beginning and end, while the inline report stays bounded and names what it left out. The large-file read is equally important. A shell-only flood test can pass while every other tool retains the old unbounded return path.

The saved record also gives completion and killing a durable representation. Ordinary command completion has an exit status; a deliberate kill has its own terminal meaning. Those facts belong in job metadata and events as well as in a readable report. Model-facing prose is a convenient account, not the sole source of state.

This continues Chapter 3's distinction between effect and narration. "The tests passed" is not a substitute for the captured result. "The job is done" is not a substitute for a lifecycle that can explain how it ended.

A Terminal Is More Than Two Pipes

Returning control solves only half the problem for an interactive program. The caller also needs a way to talk back.

`send_input` writes to an existing job and then observes its response using the same wait machinery. It does not launch the command again. That makes a prompt and answer possible across separate tool calls: start the program, see its prompt, send input, receive the reply.

The shell handler and the shell job have different finish lines. The handler returns once the process is running; from there the job owns the process, while a reader goroutine collects output and the eventual exit status. Waiting for exit inside the handler would leave a blocking call underneath the new handle.

Dispatch must respect the other half of that handoff: handler return is not job completion. Finishing the job closes its output file. If dispatch does that as soon as launch returns, it cuts off the reader still collecting the process's output. The result can look like successful execution with an empty report and zero saved bytes. That is not a formatting problem in the capture code. It is the wrong owner declaring the lifetime over.

For a shell job, stdin and stdout pipes seem like enough. The source's debugger experiment says otherwise. Delve refused an ordinary nonterminal invocation, emitting "Stdin is not a terminal." A fake prompt printed by a tiny helper could never establish that a real interactive debugger would accept the same setup.

The design therefore runs shell work through a pseudoterminal. A PTY gives the child terminal behavior rather than merely a pair of byte streams. Programs can detect the terminal, use their usual interactive modes, and issue prompts the model can answer. Output is a merged terminal stream, with the buffering and echo behavior that implies; it is not two pristine independent stdout and stderr logs.

The source's deterministic input check makes a subtle distinction here. Waking on the command's echo is not the same as waking on the program's reply. An echoed input line proves that bytes reached the terminal machinery. It does not prove that the program processed them. The test waits for the meaningful prompt or response, not just any comforting repetition of what was sent.

The historical debugger demonstration then reaches beyond that helper. The source records a second-edition Gemini session driving `dlv` through the tool: launching it, waiting for its prompt, setting a breakpoint, continuing, reading a variable as `42`, and quitting with exit code `0`.

The mistake came in reading that evidence. The student saw escaped text in a serialized display and concluded that the pattern contained two backslashes before each parenthesis. It also concluded that the first four calls had waited their full ten seconds, and paid for a repeat. The original log disproved both claims: the decoded pattern was `\(dlv\)`, and each of those four calls had returned in under three-tenths of a second. Quit correctly waited for exit without a pattern. The repeat worked, but it answered a question the retained evidence had already settled. Read the decoded argument and measured intervals before diagnosing behavior from its printed representation.

That is a good reason for a live demonstration: it answers a concrete question a toy cannot. It does not establish every future debugger version, terminal mode, or operating system. The chapter records PTY measurements on macOS and treats Linux behavior as assumed, not measured. Portability should not acquire a checkmark merely because the chosen library is portable in principle.

A Directory Is Not a Shell Session

A PTY does not mean every command shares one persistent shell. Each `run_command` starts a fresh shell job. That isolation matters when two commands overlap: one bash cannot serve as two independently executing jobs with unrelated working directories and state.

Persistence is still available, explicitly. A shell can itself be a long-running job, with later `send_input` calls changing its directory and running commands inside it. Its handle identifies the session, and the ordered input events preserve what happened there. The source's distinction is not state versus no state. It is attributable state versus an ambient shell whose

history silently affects the next call. You can compose a persistent session from isolated jobs; you cannot recover isolation merely by wishing a shared shell had forgotten something.

The source reports that roughly seven in ten archived command calls began with `cd`, mostly to the same few directories. That looked like a demand for persistence until the measurements separated directory changes from environment setup, which was rare. The agent was repeatedly correcting where it had been launched, not rebuilding a rich shell session. The remedy was a sensible launch workspace plus a `cwd` argument for an individual call.

A relative `cwd` resolves against that workspace. A missing directory is an error, not an invitation to fall back silently and run the command somewhere else. The effective directory belongs in the job's account and appears in the report when it differs from the default. The serialized `tool.job.cwd` field records the optional override, however, rather than repeating the default absolute path on every job. Absence of that override does not mean execution has no working directory.

The workspace and the directory holding logs and saved output are separate configuration. Commands may need to run in a project while their event log lives elsewhere. Deriving one location casually from the other makes moving evidence storage change program behavior—a remarkably expensive interpretation of "keep the logs over here."

There is deliberately no sticky `set_cwd`. A directory change made dozens of calls ago can disappear from compacted context while continuing to govern destructive commands. Making that safe would require keeping the setting permanently visible to the reducer. A one-call override has no such afterlife. The wrong wake costs attention or time; the wrong directory can remove the wrong build tree. Both knobs stay local to the call rather than acquiring invisible persistence.

Killing Has to Mean Something

Once work can persist across calls, stopping it becomes an ordinary operation rather than an emergency trip to Activity Monitor.

But a shell job is rarely one process. The shell can start children, which can start children. Killing only the shell PID may leave the test runner or browser happily alive. The chapter uses process-group control so cancellation reaches the shell's group rather than just its first process.

That is a practical cleanup boundary, not a sandbox. Process groups do not constrain filesystem access, network access, or credentials. Nor do they prove that no descendant can escape the group by creating a new session or group. Chapter 3's shell still has the authority of the launched process; Chapter 4 makes a defined set of that work more manageable. Supervision is not confinement with a friendlier name.

There are two truths to preserve when a kill happens. The external work must be stopped as far as the control mechanism promises, and the job must become terminal as *killed*. A late completion must not overwrite that state with an ordinary success or failure.

The source's audit caught a wonderfully dishonest-looking report produced without any need for a dishonest model. The process was killed, but the finishing goroutine marked the job `done` with `exit_code -1`. The report included both the ordinary completion claim and a note saying the job had been killed. The grader searched for the word "killed," found it, and awarded the point.

The failure explained itself in the very text that satisfied the check.

The corrected observation checks structured state and rejects a post-kill report that calls the job done or presents an ordinary exit code. A killed process may of course produce a low-level wait result in the operating system. That does not make the framework's public outcome "completed command, exit -1." The reason the operation ended is part of its result.

The late-result rule matters for non-shell work too. A file read can remain blocked below the job layer and return after cancellation. Marking the job killed cannot necessarily interrupt every underlying system call or foreign handler. It can, and must, prevent the eventual return from resurrecting the job as successfully finished. Logical lifecycle control and physically stopping an operation are related promises, not identical ones.

Killing must also wake observers. A caller waiting thirty seconds should not have to wait out the remaining interval after another caller kills the job. Terminal state is already available. Notification is part of the state transition, not an optional optimization for impatient callers.

Shutdown exercises the same machinery with a different reason. Outstanding jobs are killed because their owner is exiting, admission is closed, and cleanup is waited for rather than merely announced. "Cancellation requested" and "shutdown complete" are different claims; returning from shutdown should not leave its caller guessing which one the application means.

A live test in the historical account missed the relevant observation: its sleep had expired before EOF, so it proved nothing about shutdown killing. The bounded repeat first established that a longer-running process was still alive, then sent EOF, then observed it gone and inspected the shutdown reason. To show that an action stopped something, the something has to be running when the action occurs. Even sleep can trick a demonstration if nobody checks the clock.

Concurrency Tests Need Events, Not Hope

The races in this chapter are small enough to fit in a few lines and large enough to invalidate the architecture.

Consider a test that starts a waiter goroutine and immediately kills the job. If the waiter has not actually begun waiting, it can simply observe the already-terminal state. The test passes even if killing fails to notify existing waiters. In the second-edition review recorded by the source, removing notification still passed the initial test twenty times.

Twenty passes did not make the ordering more real.

The corrected test establishes that the waiter has reached the relevant point before killing. Similarly, releasing a blocked file read does not prove its late result has reached the job's finish path. An assertion made too early can miss the overwrite it was meant to catch. The test needs evidence of the transition, not a plausible amount of sleep after releasing it.

Even the repair had a failure path: its first version raced during cleanup when the intended assertion failed. A buffered result channel let both correct and defective executions finish cleanly. The intended assertion appearing in the output did not cancel the race warning beside it.

This is the difference between designing a test around a schedule and designing one around an ordering. A sleep suggests that the scheduler has probably done something by now. A synchronization point establishes that the thing relevant to the claim has happened. Timing remains useful for testing a wait interval, but it cannot stand in for evidence that a goroutine entered the state whose notification is under test.

The source applies the same skepticism to its larger checks. A mutation must actually change the intended behavior before a test failure counts as detection. A mutated program that cannot build because its PTY dependency was omitted demonstrates broken test setup, not a protected lifecycle. A missing debugger that causes a skipped check is an untested mechanism, not a passing interactive path.

Even a legitimate compatibility failure can have a surprising cause. The inherited Chapter 3 fixtures assumed commands returned synchronously. Under the new design, a cold `go run` that exceeds the default wake can correctly produce a running report where the old fixture expected an exit code. That is not evidence that the process was cancelled or the wake is broken. The fixture's timing assumption has become visible.

The account keeps historical scores and failures separate from later corrected checks. A perfect score from checks unable to distinguish shell-only supervision would not prove a dispatch-wide job model. Nor do accepted student checkpoints with failing results become clean passes because the chapter moved on. A score describes what a particular observer noticed. The underlying claim still needs an observer capable of disagreeing.

Take the Library Outside Its Own House

The chapter has a second architectural test: a small application in a separate Go module imports the public library and registers a custom tool.

This is not a second loop. The consumer creates an Ensemble and Agent through the public API, adds a declaration and handler, asks a question, and shuts down through the public lifecycle. Its tool enters the same dispatch and job machinery as

the builtins. If the consumer has to import internal packages, assemble a private scheduler, or copy the conversation loop, the public boundary has not actually carried the application.

A separate module makes that failure harder to hide. Code inside the repository can casually depend on internals that a real importer cannot reach. Building the consumer independently asks a narrower and more useful question than whether the core CLI still compiles: can an outsider compose this capability through the interface we claim to offer?

The tool itself need not be glamorous. The source uses `shout`, which derives an uppercase result from its argument. Its simplicity helps isolate the framework path. A declaration with a name, description, and object schema must reach the provider; the requested handler must execute; its successful result must return under the same call ID; and its output and job metadata must be saved. A final answer saying "HELLO FRAMEWORK" proves little if the model could invent it, use a shell, or recover from an unknown-tool error.

Normal registration adds a tool to the Agent's existing registry rather than replacing its builtins. The historical accepted consumer disabled builtins specifically for its demonstration, removing the shell as an alternate way to produce the same answer. Those are different choices for different purposes: an extensible library should preserve the caller's selected capabilities, while a focused demonstration can deliberately narrow them to make the observed path unambiguous.

The source records successful consumer runs through all three providers, with the exact derived output stored in the job and correlated back to the call. These are historical observations, not runs performed for this companion. They support the external-use path exercised then; they do not make every current public API spelling or every internal ownership relationship self-proving.

One of the first successful runs also lost its event capture. The launcher mistook a log header for an event and deleted scratch too early. The model call had succeeded; the evidence collection had not. Repairing capture required a repeat. Saving the raw events before parsing them would have made that parsing mistake cheap.

The Logger Follows the Parents

The external tool also emits a timestamped diagnostic. That tiny act tests a larger ownership decision.

The original refactor story, carried into the source, reached thousands of lines of Go without a logger and then introduced one. A logger is wanted nearly everywhere. That makes it an excellent temptation to add a global, or to pass another little service bundle through every function until nobody remembers who owns it.

Ensemble already supplies the root of the application. It owns the logger. Agent retains its Ensemble, Engine retains its Agent, and the executing tool retains access to its actual owner. The diagnostic follows that chain to a narrow `Logf` interface. The caller does not need the concrete logger, another copy of it, or a direct import of the sibling package that happens to implement it.

“Actual” is doing work here. Constructing a convenient substitute Agent with the same logger field would make the message print while breaking the ownership account. So would a mutable global that happened to point at the right writer during the test. With multiple applications or Agents in one process, those conveniences become cross-talk.

The source keeps the star topology from the earlier architecture: shared vocabulary belongs in the common hub; behavior stays with its responsible package. An acyclic sibling

import can still violate that design. Go's compiler proves absence of import cycles, not fidelity to ownership. Where the receiver rule prevents a method from living beside the responsible behavior, a free function is preferable to moving the behavior into common merely to make method syntax possible.

The standard logger provides synchronized writes and timestamps behind the narrow interface. There is no need to invent a logging subsystem to prove this point. The interesting result is that a leaf unknown to the framework can reach the application's existing facility through its parents.

Still, the printed line establishes only that a line reached a writer. Source inspection has to establish the path. The source records an earlier logger check rejecting a valid multi-hop path because it expected a particular field selector, and a globals check relying on variable names. Neither spelling preference could settle architectural ownership. Runtime evidence and source review answer different questions; making one impersonate the other produces very confident nonsense.

Responsive to Work, Still Deaf to You

The chapter's model now has a useful rhythm. Launch a long test, receive a running report, inspect another file, wait for new output, collect the eventual result. Start a server without requiring it to terminate. Talk to a debugger through a real terminal. Stop a job without destroying the conversation that explained why it was started.

The second-edition live account records delayed jobs, prompt-and-response exchanges, and explicit kills through three providers. One session also demonstrated a smaller inline cap while retaining more bytes on disk. Those observations show the mechanisms composing in real exchanges. They do not replace

the targeted ordering tests, establish unmeasured platform behavior, or prove that every foreign handler can be forcibly interrupted.

Four control tools and one ordinary-job path are enough for this stage. There is no external tool server connected yet, even though a tool the framework does not own motivates the design. There is no new scheduler in the consumer. There is no mailbox or interrupted-turn state hiding inside job supervision.

That last omission is something a user can feel. Start the long test, then type “Stop, wrong directory.” The work no longer holds the agent forever, but the foreground loop can still be parked in its current wait. The input may not be noticed until the wake returns. A handle makes the work addressable; it does not make the human’s message arrive sooner.

Chapter 5 changes that foreground shape: prompts, interruptions, and job completions need a way into a loop that does not block on a tool. It is a separate problem, not a reason to bolt more meanings onto a wait timer here.

Chapter 5: Two Seams and a Loop

The test is still running. You type, “Wait, wrong directory.” The application accepts the line. Nothing happens.

Chapter 4 made the work survivable: the tool has a job handle, its output is retained, and the end of a wait does not kill it. But the conversation loop can still be parked waiting for that job’s report. The goroutine that ought to hear you is the goroutine taking a nap. Moving execution off the foreground was necessary; it did not move *observation* off the foreground.

The source chapter pinpoints the blockage: the tool already runs on its own goroutine, but the conversation loop parks in a `select` awaiting its report. That same loop must drain inbound events. Hints can reach the channel and still sit unread because the drainer is parked.

The repair is not a shorter timeout. It is a change of ownership. The Agent owns a conversation loop that stays available for incoming facts. HTTP and job observations happen elsewhere and return through the same entrance as human input. The application can watch that conversation without borrowing its steering wheel.

One Owner, Several Messengers

The tempting design is to let each source of activity do a little conversation management. The stdin reader can append a hint. The HTTP worker can append a response. The tool worker can

update context when it finishes. The GUI can inspect the current turn and decide what the next line means.

Now everyone has a small, reasonable responsibility and a slightly different idea of the present.

The actor design makes one goroutine responsible for the Agent's conversation state. Its mailbox accepts prompts, hints, interrupts, request submissions, and private worker outcomes. The actor drains those messages, records and reduces events, decides what happens next, and notifies observers. Workers bring facts back; they do not append directly to live History or mutate the Engine's context behind its back.

The queue need not be exotic. The source uses a mutex-protected slice plus a buffered wake channel with room for one signal. Posting appends under the mutex and attempts to signal. If a signal is already pending, another is unnecessary: the queued messages, not the channel's contents, are the work. Draining takes the accumulated slice and clears it.

That distinction buys a simple, useful guarantee. A producer need not wait for the actor to become a receiver. This is not lock-free—enqueueing briefly takes the mutex—but it avoids turning a busy actor into a blocked stdin reader. Nor does a full wake channel mean a message may be dropped. The queue has already retained it. Reliable admission matters particularly for requests: a caller cannot receive a reply to a request the actor never got.

One FIFO gives the owner one order to process. Separate channels for hints and completions would let a `select` choose arbitrarily when both were ready. That would introduce a new scheduling decision after admission, precisely where the application wants to preserve which fact arrived first. The queue does not discover a universal order among concurrent events; it preserves the order in which this Agent accepts them.

The actor must also stay out of the HTTP call. Otherwise the first slow provider request recreates the deafness with a different logo. It records the request boundary before releasing a transport worker, then returns to its mailbox. That boundary tells later reduction which input could have influenced the request and which input arrived too late to be in its bytes.

Tools require another distinction. The execution worker runs the handler. A separate report worker observes the job until completion, a pattern, or the callback delay. Combining them would make a blocked handler prevent the running-job report that was supposed to rescue the caller. A hint can wake the report worker without stopping the execution worker.

This is not a promise to use exactly two goroutines. Processes need readers; transport needs workers; jobs synchronize their own output and lifecycle. The useful count is the number of *owners*. One owner orders the conversation. The job subsystem owns job state. Messages cross those boundaries instead of shared mutable guesses.

The source's earlier concurrency history makes that point concrete: the recurring fault was a parent reading stale child facts while the child's actor wrote them. Adding more careful-looking reads did not establish ownership. Moving the read onto the owning goroutine, or communicating through the mailbox, did.

A Hint Has to Reach the Model

A mailbox acknowledgment is not the finish line. "We received your message" can be perfectly true while the model remains oblivious for an hour.

Suppose the model starts a test and chooses a callback delay of 3,600 seconds. A human then says, "Check the first failure before changing anything." The actor can now record that hint

promptly. It must also wake the foreground job observation promptly. Otherwise the queue is responsive and the conversation is not.

The wake ends the observation, not the test and not the turn. The outstanding tool call gets one correlated return under its original call ID: the existing handle, bounded unseen output, and the state actually observed. If the test is still running, the return says so. If completion won the race, the return reports the real completed outcome. “Observation interrupted; job still running” is an entirely coherent result. “The test failed because somebody spoke” is not.

The next model request includes both that return and the incoming hint. The call is paired, the new instruction is visible, and the model can decide what to do while the original job continues. It need not wait for the test to finish before answering a question or issuing fresh calls.

This applies equally to a later `wait_for_job`. Waking it does not create replacement work. A subsequent wait observes the same handle using the new call’s limits. Available unseen output, or a retained finished result, can return immediately. The job still owns its output file and cursor; ending a wait neither discards bytes nor entitles the next wait to repeat bytes already consumed.

An addressed question is more demanding than a notification that somebody wants attention. Its sender, recipient, and request or turn correlation must survive delivery along with the actual question. Without that information, the model may be awake but unable to answer the right caller. Such a message is reliable actor input, not a progress cue that an observer may lose.

Bill reported the operational version: student agents were calling a parent stuck in `join_agents`; the calls timed out without answers. The parent was waiting for the students to

finish, while the students needed the parent to proceed. A longer timeout only gives the stalemate more comfortable seating.

The remedy is the same observation boundary. Wake the parent's foreground wait, preserve the original join job, deliver the question, and later resume observation of that same join handle. Bill's report motivates the design; it is not a captured wire trace establishing the incident's exact timing. This chapter demonstrates the seam with an application-registered join-like handler, not with a new child-agent framework. Later supervision tools can inherit the mechanism without being invented here.

There is a second place where a hint can disappear: an HTTP response that contains only final assistant text. The hint arrived too late to alter the request already sent, but that does not make it somebody else's problem. Preserve the assistant reply, apply the pending hint, and continue the *same* turn with another request. The source records an initial implementation that instead returned the final reply, leaving the admitted hint waiting for a later `Ask`.

That implementation had received the message. It had not delivered the message in the sense the human needed. Responsiveness ends at useful model input, not at a cheerful acknowledgment.

The Rest of the Batch Is Still Real

A model response can ask for several tools. Dispatch remains serial within that batch. A wakeup may therefore find one call finished, one job running, and several calls not yet dispatched.

Sending the hint straight to the provider at that point would leave dangling calls. Finishing every queued call first could hide the hint behind another long wait. Replaying the whole batch

would duplicate effects. The right answer is less magical: account for each call according to what actually happened.

Completed calls keep their real results. The dispatched job gets the truthful observation result and its existing handle. Every undispatched call gets a correlated error saying it was interrupted and not executed. Those handlers do not run, have no effects, and allocate no job IDs. The original assistant calls and signatures remain intact; the framework supplies honest results rather than editing the model's past to make the present convenient.

Only then can the next request carry the incoming message. The discarded tail does not replay automatically. If the model still wants one of those operations, it can ask again with the new information in view.

The dispatch boundary decides the close race. Once a call has started, it belongs to the dispatched case. It cannot be called "not executed" merely because its handler has not returned. That distinction is especially important for effects: a write can happen before the result arrives.

Chapter 4's literal-next-call rule for `tool_limits` survives this interruption. Pending limits are consumed by the next call even if that call receives a not-executed error, then reset. An undispatched `tool_limits` cannot arm new settings. Without those rules, a setting intended for the discarded tail could quietly govern fresh calls after the hint. The user would have corrected the plan while an invisible piece of the old plan remained armed.

This is one reason not to implement wakeup as a convenient return from whichever nested function happens to notice it. The conversation owner has bookkeeping to finish across the entire response batch. Early control return is useful only if the history remains a valid account of the work.

The timer needs a finish line too. A message and completion can arrive together; both facts must survive, but each outstanding observation call still receives only one return. Retire the old timer

when that observation ends. Otherwise it can wake later and manufacture a ghost result during a new wait.

Output notifications have a narrower meaning: recheck the observation's conditions. They do not necessarily mean return immediately. The source records a wakeup implementation that returned on every new byte and made `send_input` report a PTY's echo before the process exited. Seeing your own input echoed is not seeing the command finish. Unless an actual hint or question wakes it, `send_input` retains its pattern, completion, or callback-delay conditions. The immediate read of available bytes on a resumed `wait_for_job` is a different operation, not a universal rule for all observers.

Concurrency has a talent for making two reasonable sentences sound equivalent until a terminal repeats your command back to you.

Four Different Ways to Stop

A hint, an interrupt, a job kill, and shutdown all disturb waiting. They should not share one vague cancellation policy.

A hint changes the pending conversation context and may end the foreground observation. It does not change the turn state or end the active turn. Explicit live input is classified by the Agent using its current reduced state: idle input starts a turn; input during an in-flight or tools-pending turn becomes a hint. The stdin reader must not classify from a snapshot that may already be stale.

An interrupt ends the active turn and cancels its in-flight model request. It does not end the actor. The actor keeps draining messages, records later results, and can accept another turn. `Interrupted` is a state rather than a side flag so that a late result cannot accidentally resume the old turn's dispatch.

The unresolved calls still need honest pairing. A wait was interrupted; its job may remain alive. Undispatched calls were not executed. A later real result must not become a duplicate vendor return, nor finish whichever newer turn happens to be active. The accepted implementation retains that later fact as `job_report`, with the bytes in the job's output file. Old work is still evidence even after it has lost the right to advance the conversation.

An explicit job kill has different semantics. It stops managed processes through the job machinery and makes the killed lifecycle terminal. A Go-only handler may not be physically cancellable; the kill can return without joining it and suppress its late output. That is not the policy for an ordinary interrupted turn, where the actual later report remains valuable.

Shutdown closes admission, cancels in-flight HTTP, stops managed processes, joins workers, records outcomes, and saves. A non-cancellable Go handler may delay shutdown because its actual outcome still has to be collected. Pretending it stopped would make shutdown quicker by making shutdown false.

The actor has to remain alive long enough for joined workers and admitted kill calls to deliver terminal facts. A shell handler returning after launch is not the process finishing; Chapter 4 already taught that distinction. It matters again at the final save. Repeated shutdown is safe, and later requests receive a distinguishable stopped error rather than slipping into an abandoned queue.

The historical human-CLI demonstration shows the useful separation. The source records a hint acknowledged during a running shell job, an interrupt, and a new turn answering while the old job remained alive. Its real late output arrived afterward. That is evidence of an interface usable before old work finished, not evidence that interruption had killed the work.

The same record preserves a metadata correction: the initial implementation emitted `job_finished` with status `running` when the shell handler returned from launch. The late output was genuine; the lifecycle label was wrong. Correcting the terminal-state guard did not require pretending the original demonstration had measured something different.

Synchronous Does Not Mean a Second Loop

An embedding application may still want the simplest possible call: ask a question, receive its answer. The public API need not become a tangle of channels just because the implementation becomes an actor.

`Submit` posts a separate request and returns its private, one-element buffered reply channel. `Ask` waits on that request's reply, caller cancellation, or actor shutdown. The actor keeps working independently. The buffer lets it deliver a completed result even when the caller has stopped waiting.

"Private" is the important word. Waiting for a shared idle state and then reading the Agent's last text would not identify which request completed. Two callers could receive the same answer, or one could receive the other's error. Reliable request completion carries that request's text and error. Shared progress cannot impersonate it.

Blocking `Ask` and `Submit` deliberately queue separate requests rather than turning into live hints. This preserves a useful distinction between "answer this next" and "adjust what you are doing now." The legacy stdin form with a `user` key and no `kind` goes through that request/reply adapter. Explicit live

prompt and hint forms use current turn state. Compatibility is preserved by routing old behavior through the new owner, not by keeping the old orchestration loop hidden nearby.

When persistence is configured, the actor saves before replying. A failed save is a failed completion, not a durable success with a logging footnote. An empty `LogPath` leaves persistence to the embedding caller; it does not silently promise a saved conversation. Observers can be notified of the outcome without their delivery becoming a prerequisite for the caller's reply.

The Agent starts its actor lazily and idempotently. Repeated calls use the same owned mailbox consumer. Engine retains one-request rendering and transport; History retains reduction and persistence; Agent orders the multi-round conversation and runtime context changes. The old `Engine.Ask` orchestration and sibling loops must go after callers migrate. Two implementations that happen to agree today are two policies waiting to disagree tomorrow.

The tool-round limit belongs to this same owner. A positive N permits N dispatched tool batches in one human turn. Zero selects the named default of 200, not infinity. The effective setting is read once at turn start. After the last allowed batch, the model still gets to consume its results and finish; only a request for another batch triggers explicit not-executed results for every call and a distinguishable limit error. No dangling calls, no rewriting signed assistant content, and no permanent ban on a later turn.

A synchronous facade is perfectly compatible with responsive internals. The duplication to remove is not the waiting caller. It is the second authority over what a turn means.

The Witness Is Not the Driver

The outbound seam has almost the opposite reliability needs from the mailbox.

Before this change, a caller can ask and inspect the finished History. It cannot conveniently watch state changes, tool activity, or output taking shape. An observer receives typed observations as the actor commits facts. It is told what happened; it does not decide the next transition.

The boundary is deliberately after reduction: append the event, update state, then notify. If the actor announces a tool call before committing it, an observer can see a future that History does not yet contain. A crash between the two would leave an observation without its supposed fact. Post-reduction notification keeps the account in the right direction: observations report stored facts rather than promising impending ones.

The public vocabulary is small. `StateChanged` carries the old and new states. `PartDelta` carries a part identity and incremental text. `PartFinal` carries the complete part and its committed sequence. The core seals the observation interface so the framework owns this vocabulary and consumers can handle its defined variants. Application-level workflow markers need not become new core variants merely because an application wants to print them.

A delta has no committed-event sequence number. It is a transient preview of an open part, not independently replayable history. The finalizer ties that part ID to the committed part.

The vocabulary is ready for streaming, but the provider seam here still returns a complete response. The implementation emits a whole-part delta followed by a finalizer. A type called `PartDelta` is not proof that live tokens are arriving.

No observation gives the recipient a mutable pointer into live History. That prohibition includes more than the top-level struct. A value copy can still share a slice backing array or a nested metadata map. If an observer edits its snapshot and changes the next provider request, the witness has acquired a steering wheel by aliasing. Immutable or deeply isolated views preserve the single-owner argument across nested data, not just at the function signature.

There is an unavoidable performance bargain. Registered observer methods run synchronously on the actor goroutine, so they must return promptly. A logger that needs disk work or a UI that needs its own event loop should copy the observation and enqueue it privately. A buffered-channel adapter uses a nonblocking send and drops when full. It must not block the actor to keep every progress cue.

That means a slow display can miss progress. The alternative is a slow display freezing conversation execution. The design chooses an authoritative event log and a reliable request reply over an infallible progress feed. Observation channels are lossy cues; readers use history and independent snapshots for facts that must survive loss. A callback registered as an observer is legitimate behavior at an explicit boundary, not a function-pointer escape hatch for a broken package dependency.

“Notify me” and “make my receipt a condition of your progress” are very different requests. Only the first belongs in this seam.

Watching Is Not Collecting

The same distinction governs `Wait` and `WaitAny`.

An Agent's `Wait` consumes observations until one satisfies a predicate or its caller's context ends. It is not a replaying subscription that delivers every event to every waiter. Ensemble's `WaitAny` consumes a matching observation from a merged progress stream, with the source Agent attached. Each Agent keeps its own mailbox and conversation owner; Ensemble supplies the identity envelope through the existing parent relationship.

That is enough to watch several Agents without opening their private histories and spelunking for clues. It is not enough to acknowledge a particular submitted request. A shared idle transition may concern different work, and a bounded progress stream may drop precisely the cue a coordinator hoped to use. Decisions that require completion must retain the relevant request replies reliably, then publish progress about the result.

The source's review history includes a useful scope correction. An earlier "one wakeup" description led a reviewer to demand grouped completion, and the student initially agreed. The actual demonstration concerned observations from distinct Agents, not a measured wake count or an API that collects a group of results. Bill withdrew the added requirement. An attractive abstraction is not automatically a missing feature.

The concrete application here has an author, editor, and reviewer. Go constructs the peer Agents with different prompts and application-specific tools. The author stores a draft and can count its words; the editor reads and replaces that stored text; the reviewer records an acceptance or rejection with notes. These are tool declarations and handlers supplied by the application, not manuscript-shaped builtins added to Ensemble.

The application advances sequentially using reliable request completions. The reviewer receives the actual edited text, not merely the editor's assertion that editing occurred. The source's historical workflow shows that difference nicely: the author's first

stored draft missed the requested word count, the count tool exposed the miss, and the author revised the stored text before the editor worked on it. The tools make the shared artifact inspectable rather than relying on eloquent claims about it.

Even this sequential workflow needs synchronized draft storage. A tool worker can outlive the turn that launched it; "we run the stages in order" is not a lock on application data. If a role is interrupted, its request returns an error instead of advancing with an unfinished draft. EOF waits for the submitted workflow before shutdown.

Tagged observations let a display distinguish the Agents. Completion records and stage markers belong to the application protocol, while each Agent keeps its own durable history and sequence space. Adding the reviewer should require application wiring, not a patch to the framework. That is the payoff of exposing composition rather than implementing one favored collaboration.

It is also the limit. These Agents do not invent their team or choose the workflow. A different collaboration still needs different Go glue. The framework is composable; the application remains hardcoded. Sub-agent spawning and its supervision contract come later.

An Attachment Must Actually Arrive

Media capability may seem like an odd companion to an actor loop. It shares the same underlying concern: accepted input is not necessarily delivered input.

A swallowed hint leaves the model answering an old question. A silently dropped attachment leaves it answering without material the programmer believes it received. In both cases,

improving the prompt can become a very elaborate attempt to repair the wrong boundary.

The model catalog therefore declares image, audio, video, and document support separately. Image support implies none of the others. An unsupported attachment fails locally with a diagnostic naming the model and media type, rather than disappearing from the request.

An unknown model has no optimistic default row. Text alone remains allowed; rendering a blob requires known support. That distinction avoids converting "we do not know this model's media capabilities" into the much broader and unsupported claim "this model cannot accept text."

A blob carries a `Ref`: a local path, a URI, or a handle. If the blob is superseded, the redacted part preserves its reference so the address survives. The address is not a promise that an external file or job output will remain available forever.

Nor is retaining a locator equivalent to rendering an attachment. Local-path rendering reads the actual file bytes into the vendor's attachment structure. URI and handle rendering remain deferred even though the reference vocabulary can represent them. This is a useful kind of future readiness: the log can retain identity without claiming an implementation the renderer does not have.

The source's media demonstration sent a local red image through a public-API client and received `red`, then separately observed local refusal of unsupported audio. Positive audio, video, and PDF paths were checked at the rendering boundary, not presented as additional live model demonstrations. The distinction matters: a correctly shaped request and a model's observed consumption are related evidence, not interchangeable evidence.

One repaired check sharpens the engineering lesson. Looking for attachment bytes somewhere in the request missed a wrong Gemini `mimeType` key. Decoding the actual attachment fields caught the error. The vendor consumes a particular structure, not a bag of reassuring substrings. Whether the input is a hint or an image, delivery has to be established at the boundary that actually consumes it.

Available Is a Property of the Whole Path

The difficult failures here often look locally correct.

The stdin reader accepted the hint. The actor recorded it. The tool ran on a worker. The timer had a legitimate deadline. The observer emitted an acknowledgment. Every sentence can be true while the model still cannot act on what the human said.

The useful question follows the whole path: was the message admitted reliably, ordered by the conversation owner, paired with an honest observation return, and included in the next request without waiting for unrelated work to finish? At the other end, did the caller receive its own completion after the configured save, rather than infer it from a lossy progress cue?

That also explains what convincing concurrency evidence must distinguish. A long-delay case can show that a hint wakes observation promptly, but it cannot show that an obsolete short timer stays retired during a later wait. A passing interrupt case cannot establish hint wakeup, because interrupt ends the turn. A hint acknowledgment cannot establish delivery into the next provider request. Completion-before-message, message-before-completion, and the close race all need to preserve the message and the real outcome without duplicating the tool return.

These are questions about ownership and causal order, not reasons to let an hour elapse or to commission another broad live campaign. The source's historical provider demonstrations exercised different feature slices; they were not every feature on every provider. Their value is in the particular boundary each makes visible.

Chapter 4 gave work a lifetime independent of an individual wait. Chapter 5 gives the conversation an owner independent of the work it observes. The human can speak while tools run; the model can hear without killing them; an application can watch without driving; and a synchronous caller can get its own answer without owning the loop.

Chapter 6: Streaming, or the Same Answer in Pieces

The Agent can listen while busy now. It can accept a hint, interrupt an observation without killing the job, and tell an application what happened. Ask it a difficult question, though, and the terminal still goes quiet until the entire model response arrives.

Chapter 5 prepared `PartDelta`; it did not make the provider stream. Its whole-part notifications were useful scaffolding, not evidence of words arriving early. Streaming earns that distinction by carrying pieces all the way from an open HTTP response to a display while more of the response remains unread.

The central bargain is small: change delivery without changing the answer's durable meaning. A reply can arrive in one piece or many. History still records completed facts, tools still execute complete calls, and replay still reconstructs the same conversation. The human gets to see work sooner without making unfinished work authoritative.

Let the Response Stay Open

An HTTP body is already a stream. Reading it all before parsing is a decision to hide that property. Streaming moves the parser onto the open response: the HTTP worker hands it the response, the parser recognizes arriving fragments, and those fragments travel

through the Agent's mailbox to observers. Worker messages carry the turn identity, so late progress from an interrupted request cannot decorate a newer turn.

There are three visible kinds of fragment: readable thinking, ordinary reply text, and tool-call content. The separation matters beyond colour. Reasoning must not leak into the reply, and a tool's partly written arguments must not be mistaken for executable instructions. A provider that exposes no readable reasoning does not owe the display an invented thinking stream.

The parser accumulates the complete native response as it goes. On successful completion, the same semantic decoding used for plain delivery builds the finalized parts, with canonical arguments, provenance, usage, and opaque replay material intact. The worker returns facts to the actor; it never appends directly to History. The actor records and reduces the response before publishing final-part observations.

This preserves the existing ownership seam rather than creating a special streaming conversation loop. Rendering remains Engine's work, configuration comes from its Agent parent, and transport constructs the URL and headers. Asking for a stream changes those existing request decisions. It does not require a second orchestration API.

The source chapter argues for one `Parse`, not `Parse` plus `ParseStream`. Two entry points look modest until each vendor has two definitions of a tool call, two places to interpret stop reasons, and two opportunities to lose usage. The path used in production gets repaired; the other becomes the mode selected when somebody wants fewer variables while debugging.

One method name alone cannot prevent that drift. Shared final construction does. The accepted implementation accumulates streamed provider messages before passing them through the semantic decoder used by plain responses. Framing differs; what counts as a completed answer does not.

Plain delivery also emits a delta: one whole-content notification per part, followed by its finalizer. Otherwise every observer would need both a delta renderer and a separate renderer for non-streaming responses. “A stream of length one” is useful because consumers can actually treat it that way.

Parity here means the same scripted content produces the same completed parts and usage through either delivery mode. Two independent live generations need not choose the same words. Turning streaming off is not a seed for model determinism.

Framing Is Boring Until It Eats the Ending

The streaming dialects in this chapter use server-sent events, or SSE, over ordinary HTTP. A frame has field lines, including an optional event type and one or more data lines; a blank line ends the event. A shared reader turns this framing into an event type and payload before vendor interpretation begins.

A scanner that looks for `data:` handles a pleasant demonstration. Real framing adds a few less pleasant cases. Multiple data lines belong to one event. A colon-prefixed comment may be a keepalive, not empty JSON. CRLF must still produce a separator. A completion sentinel such as `[DONE]` is not a JSON object. An event left unterminated at EOF must not simply disappear.

These faults often leave plausible output. Losing the final event may preserve all the visible prose while dropping its stop information. The resulting problem surfaces later, wearing the wrong subsystem’s name badge.

A small shared framing reader keeps those deterministic failures out of each vendor parser. Once framing is resolved, the provider-specific work can start with a real payload rather than another interpretation of line endings.

The native request still has to be valid. Anthropic and OpenAI request streaming in their bodies; Gemini uses its streaming endpoint with SSE selected. OpenAI additionally needs its usage-inclusion option. Without it, streamed usage is absent, and an accumulator that treats absence as measured zero can make every answer appear free while everything else works. Missing accounting remains missing accounting.

Streaming also does not rescue a rejected request parameter. The source's accepted OpenAI renderer uses `max_completion_tokens`, while Anthropic and Gemini retain their own native token-limit fields. A parser cannot demonstrate liveness on an answer the provider refused to generate.

A Part ID Means "This One Again"

Imagine a display receiving fragments for reasoning, reply text, and a tool call. It needs to append each fragment to the right place, then replace the provisional rendering with the completed part. The identity must survive both operations.

The first-edition brief initially proposed a fresh atomic ID for every chunk. The IDs would all have been unique. They would also have told a consumer nothing about which chunks belonged together. Forty numbered envelopes are not forty answers.

The next temptation is to use each part's position in the finished response. OpenAI exposes the flaw: tool arguments can start arriving before the parser knows whether a text part will

occupy the first final position. A caller choosing IDs from that future list is guessing about a response that has not finished existing.

The parser has the information, so it chooses the identities and repeats them at finalization. Anthropic can retain its content-block indices. The accepted OpenAI parser reserves zero for text and maps tool indices into a separate range. Gemini needs a local continuation mapping. Zero is a valid identity; positivity is not the guarantee.

Within one response, one part has one ID, different parts have different IDs, and an ID does not change content kinds. The finalizer carries the parser's ID, not a position reconstructed from the final list. That is enough to connect an approximation to the thing that replaces it without claiming the number means "where."

The scope matters. Parser IDs may restart with every response. A turn that calls a tool usually includes another model response afterward, and its part zero is a different part. A consumer needs the pair of response identity and part ID, not a turn-wide map of bare numbers.

The current protocol exposes the transition to `in_flight` before the response's deltas, allowing a connected consumer to separate responses. That transition must remain visible even if the CLI also supplies explicit response identifiers. It is a live boundary, not by itself a durable identity that a reconnecting GUI can recover. Chapter 7 still has work to do there.

There is another order to preserve at the far end. Parsing reports the completed event before its part finals; the worker retains both the facts and the identity mapping. The actor then records the response and publishes `PartFinal` from the stored parts. An observer refreshing from History sees the fact it was just told about, rather than a log one event behind.

Text and readable thinking fragments should reconstruct their own parts, grouped by response and ID. Comparing only the concatenated text of an entire turn misses broken identities completely: the words may be right while every widget is wrong. Requiring a part to arrive in several pieces also distinguishes real incremental delivery from whole-part notifications with an ambitious name.

Tool displays have a slightly different boundary. They include arriving name fragments as well as arguments, emitting each fragment once rather than repeating the accumulated name on every update. A consumer appending the ordinary chunk field must see the tool name; hiding it in supplemental metadata does not accomplish that. Plain delivery supplies the complete name and arguments once.

The raw argument fragments may retain wire whitespace, while the authoritative tool part holds canonical JSON. At that boundary, decoded argument values are the meaningful comparison. Cosmetic differences in the display must not become differences in what the tool receives.

Seeing a Call Is Not Running It

Early tool arguments are useful to a human. They show what the model is preparing to do. They are not permission to begin doing it.

A path may arrive before the rest of a write request. Opening the file at that point would let an incomplete JSON fragment cause an effect. Even a completed-looking part is not enough: dispatch waits until the response completes successfully. An error after visible fragments must not leave behind a tool operation started on speculation.

Streaming also makes the order of explanation observable. For models whose policy requires ordinary explanatory text before a tool batch, the implementation must retain whether nonblank text arrived before the first tool-call delta. Thinking does not count as that explanation. A tool-first stream followed by reassuring prose has still shown the call first.

This cannot be recovered solely from the finalized response. A decoder may arrange final text before tools even when the wire delivered them in the opposite order. Accepting that arrangement as evidence would let the model explain after announcing the action and receive credit for explaining before it.

The final-response check therefore stays, including for plain JSON, and streaming retains the earlier arrival evidence as well. A tool-first stream or reasoning-only prefix gets corrective tool results instead of execution when the explanation policy applies. A text-first stream may dispatch after successful completion. The streaming display makes more of the process visible; it does not relax the existing gate on effects.

Local Differences, Not Imaginary Uniformity

A common observer vocabulary does not mean the providers send the same thing.

Anthropic identifies content blocks and distinguishes text, thinking, and input-JSON deltas. OpenAI has its own indexed tool fragments and accounting option. Gemini sends response-shaped frames with new content rather than the same named delta events. It has no equivalent block indices, so its parser must infer continuation from the shape and position of what arrived before.

For Gemini, a new frame is not automatically a new part. Conversely, two text entries explicitly present in one frame's parts array are not automatically one paragraph. The first entry of a later frame may continue a compatible unsigned part. That inference must stop at a signature boundary.

A trailing signature can finish unsigned text, thought, or tool content. Once attached, it binds to that exact content. Appending later words would change what it signed; replacing it would discard the original binding. A signature-only frame is therefore meaningful even when the display has nothing new to print. Ordinary text can carry signatures too.

The accumulator must preserve those distinctions: standalone signature material stays standalone, signed text stays separate from later text, and signed thought does not absorb a separate opaque part. Canonical parts retain the native payload needed for exact-source replay. Treating everything with no visible words as empty is a tidy way to break the next request.

In the reference implementation and observations, Gemini's function arguments arrived as complete objects even while its text streamed. The correct display response was one whole tool-object chunk, not manufactured subdivisions to make the stream look busier. The actor and terminal did not need a Gemini exception; the parser expressed what the wire actually supplied through the common vocabulary.

The source retains a historical capability bitmask distinguishing text, thinking, and tool-argument streaming. The accepted restart uses a simpler SSE/plain/unknown capability, with native parsers handling the kinds of chunks available. These are different representations of the same obligation, not two simultaneous requirements. Neither promises that every model exposes every kind of content incrementally.

Capability rows also age. A table update can enable a dialect the translator already understands. It cannot teach a parser a new wire format. The design contains provider differences inside translation; it does not establish that providers are interchangeable.

An Unknown Model Can Still Answer

The capability table's conservative fallback is plain delivery. An unlisted model is not a reason to reject an otherwise valid text request.

This deliberately differs from Chapter 5's media refusal. Guessing about an attachment can remove content the model needs while leaving a plausible answer behind. Falling back from streaming preserves the content and gives up early visibility. One risks changing the conversation; the other changes how long the terminal stays blank.

Model names form an open set, including test models and deployments newer than the table. Requiring every name to be known before any request can proceed would make the capability catalog an accidental access-control list. Unknown streaming capability means do not ask for streaming, not do not work.

The explicit opt-out is `DisableStreaming`. The negative name buys the intended Go zero value: a hand-built configuration streams when the model's capabilities permit it, unless the caller disables that behavior. A positive `Stream` boolean would default to false unless every caller used a constructor. In a public library, callers are allowed to construct structs. A default that works only through the preferred entrance is a suggestion.

Three Accounts of the Same Response

History, the wire trace, and the display answer different questions.

History records finalized conversation facts. `ResponseEnded` still carries the complete response, including its parts, provenance, and usage. Deltas never become additional conversation events. Persisting them as content would either duplicate the final parts or require replay to assemble a second representation of the answer. The existing log remains enough to reconstruct the live context.

Chronology in that log need not be literal message order in a vendor request. A busy hint can arrive between an assistant's tool calls and their results. OpenAI requires the paired tool results immediately after the assistant call message, so its renderer moves the complete result group ahead of the hint in the disposable wire projection. History keeps the hint where it arrived. Rendering may rearrange real, complete results to satisfy the API; it may not invent results or rewrite the historical facts.

The root-owned `api.log` answers a different question: what payloads passed through translation? The first-edition engine logged response JSON after reading the body. Streaming removed that convenient whole-body moment. Without replacement plumbing, the debugging trace would have lost its response half precisely when responses became harder to inspect.

Frame notifications restore that visibility as payloads arrive, in order, in both modes. They are not a byte-for-byte HTTP capture: SSE comments and framing are not the payload, and authentication headers do not belong in the trace. The accepted Engine redacts a reflected configured key before logging or forwarding a frame and excludes credential-bearing URLs.

Logging follows the existing parent chain to Ensemble, which owns and serializes the writer; parser callbacks carry output rather than a collection of borrowed logging services.

Observers provide immediate, provisional visibility. A display can append chunks quickly and later render the authoritative part under the same identity. It need not pretend its temporary rendering was complete. Tool-result finalizers remain distinct from model-response finalizers, or a consumer could count job output as part of the provider's stream.

These observations still do not replace reliable request completion. A completed assistant answer comes from the request's private reply, not an acknowledgment that input was admitted and not a progress queue that may drop entries. A busy hint acknowledgment is not another completed answer. Streaming adds more things worth watching, not more authorities over whether the request finished.

The Last Buffer Can Undo Everything

The human CLI sends reply text to stdout, with thinking and tool-call content on stderr. Terminal-only styling makes those channels readable without contaminating redirected output with escape codes. A pipeline receives the reply; a human sees the surrounding activity too.

That arrangement can be correct and still fail at the last inch. A buffered writer may accept every chunk immediately, then hold all the bytes until completion. The parser streams, the actor streams, the observer streams, and the human gets a blank terminal followed by the whole answer. The buffer has done an excellent job of removing the feature.

Flushing each chunk fixes that last boundary. The accepted CLI instead writes chunks directly to unbuffered stdout. Its observer queues display work without waiting for a slow terminal or dropping reply chunks. Once the Agent has joined producers, the display drains before the summary. Finalizers add separators, not a second copy of the reply.

That queue is a deliberate resource tradeoff. Optional progress may be dropped when a consumer falls behind, but a terminal promising the full reply must preserve those bytes without holding up the actor. Buffering moves the pressure into retained display work; it does not abolish pressure. Making the actor wait would recreate Chapter 5's deafness through the output side.

The source records a particularly revealing CLI failure. The initial student implementation treated any chunk beginning `hint:` as a human-hint acknowledgment and routed it to stderr. A real parser-to-actor-to-CLI test supplied assistant text in two pieces, `hint:early` and `late`. Only `late` reached stdout. The assistant had accidentally used a reserved prefix that nobody had reserved.

The correction marks actual acknowledgments by origin, using `PartDelta.Hint`, rather than inferring origin from prose. The content kinds remain thinking, text, and tool call. A fourth colour would not fix a missing distinction between who said something and what characters they used.

The corrected test also supplies the chapter's strongest liveness evidence. It withholds the rest of the HTTP response until stdout contains `hint:early`, then requires the complete reply `hint:earlylate`. The first piece must become externally visible while the response is still unfinished, or the test cannot advance. Restoring the prefix heuristic breaks that same test; removing the origin marker breaks acknowledgment routing.

Several reads after completion would prove much less. A network reader, a pipe, or a terminal can split already-finished output into many reads. End-to-end streaming is a relation between visibility and unfinished work, not a pleasing chunk count.

What the Evidence Actually Shows

The source's separate public-library application observes stream shape: delta counts by kind, finalized parts, time to first delta, and the completed answer. It uses the ordinary Agent and observer seam, rather than a private demonstration path. Reliable request completion determines when its in-memory measurements can be summarized.

Those measurements are useful for different reasons. Counts reveal whether streaming produced multiple fragments; reconstruction by kind checks that reasoning stayed separate from the reply. Grouping reveals identity errors. First-delta latency describes the initial wait. Final content and usage establish plain/streaming parity for a controlled response. None of these alone proves that a human CLI flushed before completion; that is what the gated test establishes.

A first-edition audit removed only the request's streaming flag. Final text, parts, and event log stayed identical, while the incremental-delivery check failed because too few deltas arrived. That is the chapter's thesis expressed as a fault: content correctness can survive the complete loss of early delivery.

The audit also found limits in its own checks. A per-chunk-ID mutation initially used a package-level counter and failed an unrelated prohibition on mutable globals as well as the intended identity check. It had to be rewritten to isolate the identity fault. Removing thinking deltas did not fail a reconstruction check that

examined only text. A failed check's name is not enough to establish why it failed, and one content kind's coverage does not silently cover the others.

The retained live shape pair used `gpt-4o-mini` and happened to return the same numbered text in both modes. Streaming delivered its first delta sooner but took longer overall in that small pair. That is evidence of earlier visibility in those runs, not a throughput improvement or a general latency benchmark. The model still has to produce the answer.

The human demonstrations exposed other boundaries. An Anthropic launcher watched file modification times and labeled an input as a new turn. A log rewrite changed the timestamp while the prior turn was still active; the durable record showed that the input was actually a busy hint. A changing file is evidence of a write, not of request completion.

The Gemini demonstration required a different correction. Bill had required Gemini 3.0 Flash or newer and specifically requested 3.8 Flash for hints. The student selected 2.5 from discovery, and the handoff and first review missed the restriction. The run worked, but success on the wrong model did not establish the requested behavior.

One bounded run on the actual `gemini-3.8-flash` CLI supplied the missing evidence. A real command created a waiting marker, paused for a release marker, and then printed its result. While it was waiting, a busy hint changed a requested lighthouse paragraph from red to blue and required a recognizable answer prefix. The launcher observed the actual stderr acknowledgment before releasing the command. The next provider request contained both the successful tool result and the hint; the answer used the prefix and described a blue lighthouse. Tool and CLI exited successfully.

That record establishes delivered steering, not merely accepted input. It does not establish leisurely token-by-token typing: its three nonempty stdout reads spanned only about a millisecond. The gated CLI test remains the stronger evidence that bytes become visible before response completion. No production change or repeat of the other provider demonstrations was needed to repair the model-selection gap.

Historical grading failures also need their causes kept separate. One fixture selected an unknown model while demanding streaming, contradicting the intended plain fallback. A reference run that supposedly disabled streaming still streamed. Neither result can be explained honestly by saying simply that streaming passed or failed. The source preserves those records; they are not a claim that this prose revision reran them.

Enough Visibility to Want a Better View

The terminal now exposes reasoning, reply text, and half-written arguments across two file descriptors. It appends rather than redrawing completed parts. That is useful development tooling, and not much of a layout engine.

The next interface can do better without changing who owns the conversation. It has kinds for routing, response-local part IDs for grouping, and finalizers that replace approximations with stored facts. It still needs durable widget identity across reconnection, connection lifetime, and a sensible arrangement of the content. Those are GUI problems, not reasons to pull display policy into the parser.

Chapter 7: Everything Is an Artifact

Chapter 6 gets the words onto the terminal before the answer is finished. That is a considerable improvement over staring at nothing. It is less helpful when the thing worth noticing has already scrolled away, the tool arguments wrap across six lines, and the human is trying to compose a correction while the next tool starts.

The GUI has a harder job than making that terminal attractive. It must give the human a usable view of ongoing work, recover that view after a disconnect, and leave enough room to intervene. A display that announces an execution which never happened is misleading. A pause button that prevents interruption is worse than no pause button. Speech that continues reading yesterday's history is merely thorough in the wrong direction.

The source chapter builds one reusable scroll component, not the whole workbench. Its central choice is to keep the Agent unaware of the GUI. Observation and control cross public interfaces; layout and sockets stay outside. The interesting work lies in making that separation hold when the browser is slow, absent, speaking, reconnecting, or wrong.

One Agent, Several Receivers

The source's radio analogy is useful: a station does not stop transmitting when somebody unplugs a receiver. A browser subscribes to observations from the existing Agent. A second browser subscribes to the same Agent. Closing either tab changes neither the conversation nor the running work.

Unlike a radio, the browser also gets a recent recording and can send input back. The recording is durable History, not an accumulation of whatever notifications a particular connection happened to catch. The return path is the same public `Post` entrance the terminal uses. A message arriving while the Agent is busy becomes a hint under the actor's existing rules; the browser does not choose a turn for it or start another model loop.

The application constructs one Ensemble and one Agent for terminal and browser alike. The GUI and WebSocket server live in an optional Go module using the public library. Core acquires no WebSocket dependency, rendering package, or display configuration. Configuration remains with Agent, Engine reaches it through its actual parent, and the transport owns sockets. There is no need to make a web server the Agent's parent merely because it knows where a button is.

That boundary has operational consequences. The accepted server listens on localhost, serves the page and `/ws`, and takes its asset location separately from the tool workspace. Its public flags override inherited provider environment defaults; credentials remain in the intended process environment, not command arguments or logs. The working directory supplies the default workspace and log location. Terminal EOF detaches that input rather than shutting down the browser. These details preserve one running application with several entrances, rather than letting each entrance quietly invent its own lifecycle.

A Requested Call Is Not a Started Tool

Streaming already shows the model composing a tool call. The GUI also needs to distinguish that request from execution, and execution from a returned observation of the job.

`ToolDispatched` supplies the call ID, tool name, and raw arguments at the execution boundary. `ToolFinished` supplies the call ID, bounded result text, and error flag when the tool wait returns. That result may contain a handle for a job still running. The full output remains recoverable, and later real job reports must remain visible even if the human has interrupted the turn. “Finished” describes the wait reported here, not a universal claim that the process is dead.

`TurnEnded` gives displays the final reply, excluding intermediate tool narration, or a safe diagnostic when the turn fails or is interrupted. It is progress, not the reliable completion used by `Ask`. A missing display notification cannot be allowed to redefine whether a request completed. Ensemble identifies the source Agent when broadcasting these observations; tools do not need a copied Agent identity or a GUI parent.

The source records a revealing Edition 3 defect: the GUI announced tool starts for a batch the core had refused. It did so live and again on replay. The requested calls existed, and refusal results existed, so the display inferred the missing action between them. It had produced a convincing account of an execution that never occurred.

The repair belongs at admission, not in a more persuasive card label. Saving and publishing the dispatch fact happen inside the same boundary that releases the handler. A refused call remains visible as a request with its refusal result, but receives no dispatch announcement. Cancellation after publication can stop admitted work; publication must not describe a reservation that was never released.

This is why the three vocabularies stay distinct. Stored events such as `tool_called` and `tool_returned` are historical records. Mailbox reports are internal signals. Observations are broadcasts. Similar names and related content do not make them interchangeable evidence. In particular, the existence of a stored request is not proof that its handler ran.

Pause Buys the Next Boundary

The actor dispatches tools serially: start one, wait for its report, then consider the next. A continuing job can outlive that report, so this is not a promise that only one operating-system process exists at a time. It is a promise of a dispatch boundary between calls.

That boundary is what makes pause useful. Starting a whole batch in parallel would leave a pause arriving just afterward with nothing left to prevent. Serial dispatch gives the human a place to catch the next call. An engineer considering parallelization needs to see that dependency, not just a loop that appears insufficiently ambitious.

Pause is live Agent control, not another conversation event awaiting reduction. The Agent owns the gate; the browser reaches its public `Pause` and `Unpause` methods through the server. There is no second GUI gate and no Engine sibling handed to the transport.

The accepted implementation uses a mutex and a wake channel. When the gate is closed, a small boundary waiter waits for unpause or cancellation and reports back to the actor. The actor continues consuming its mailbox. Blocking the actor itself would strand the interrupt intended to rescue the human from the pause.

The two wake reasons mean different things. Unpause permits the next dispatch. Cancellation ends the current turn, leaving the Agent usable for another prompt and other jobs free to report their actual outcomes. The gate deliberately bypasses mailbox FIFO order because a pause waiting behind pending reports may arrive too late to hold the next start.

None of this retracts an admission already made. A running tool continues producing output while paused. Pause is neither process suspension nor retroactive cancellation. Its power is narrower and easier to reason about: no new tool starts past a closed boundary.

Reconnecting Without Reconstructing the Conversation

A fresh browser and a reconnecting browser use the same subscription path. Neither supplies a cursor or sequence number. The server sends the recent durable history, then accumulated content for active parts, then live delivery. The browser receives identities and renderable payloads, not provider SSE frames to interpret for itself.

The history window is fixed: the last hour or the last 100 renderable events, whichever reaches farther back. A busy hour can contain more than 100 events; a quiet Agent can supply older context to reach 100. An event producing several rendered parts still counts as one event. Counting socket messages or unique cards instead would silently change the policy.

The subscription also reports current turn and pause state, marked as replay. The turn snapshot uses `state_changed` with `from` and `to` both set to the current state. Equal endpoints say "this is where things stand," without inventing a transition. The

initial subscription snapshot omitted `from`; the source records the corrected subscription check and a restored-defect mutation that detects the omission. Likewise, an empty delta still carries `chunk` with an empty string. Empty content does not make a public field optional.

The hub keeps two tiers of state. Durable events supply finalized parts, user messages, calls, and results. An in-flight map accumulates the chunks of parts still streaming. Live delivery sends only the new chunk; subscription sends the prefix accumulated so far. Finalization removes that prefix, because History now supplies the authoritative content.

There is an internal frontier even though there is no browser cursor. On each observation the hub checks for new history since its captured event-log `Seq`, projects the renderable facts, and combines them with progress. The event log supplies order. A connection does not get to claim which durable facts the server has already captured.

Chapter 6's parser IDs are too short-lived for this job. They distinguish parts inside one model response; a GUI card can survive several responses and human turns. The actor therefore assigns a conversation-scoped display ID, broadcasts it through the root observer, and stores it on the final part. Replay updates the card that received the live chunks. Together, `agent_id` and `part_id` identify the logical part, even when several events update its dispatch status or job metadata.

This identity is for display, not a new interpretation of provider call IDs or signed content. Those retain their original meanings. Nor does writing IDs to the log implement process restoration. When save/load arrives later, new parts after restart must avoid colliding with old cards; this chapter recovers a browser view while the Agent continues running.

The awkward case is not a reconnect after everything is quiet. It is a reconnect exactly as a part finishes. Without coordination, the subscriber can receive completed history and an obsolete partial for the same content, or miss the final between two snapshots.

Append-only history does not solve that race. Appending can replace a Go slice's backing array while another goroutine reads it. The history owner snapshots the requested suffix under its lock. The GUI then coordinates the captured history frontier, partial map, and subscriber admission under one handoff lock. A captured final removes its partial before publication. Active partials replay in assigned creation-identity order rather than Go map iteration order. Interruption clears abandoned partials that will never finalize, without deleting completed facts or late real job output.

The source's testing history shows why the timing matters. Removing final-part cleanup initially passed: the test reconnected after `TurnEnded`, whose separate cleanup had already emptied the map. Reconnecting while tools were still running exposed the stale partial. Both tests reconnected successfully. Only one visited the interval in which the defect existed.

A Slow Browser Does Not Own the Actor

The observer callback captures history and updates partials, then offers messages to clients without waiting for network I/O. Each connection has a buffered channel and a writer goroutine. A full live channel drops that client's progress update rather than blocking the Agent.

This is a different promise from Chapter 6's terminal queue, which preserves reply bytes. A browser can miss live progress and recover durable content through subscription. The choice

protects the actor from a slow receiver; it does not guarantee every receiver sees every intermediate state.

That bargain cannot apply indiscriminately to replay. Dropping half the reconnect history would break the recovery path offered in exchange for lossy progress. The accepted server admits the complete replay as one queued batch while adding the subscriber. The connection writer sends the batch outside the lock, so history admission is coordinated without turning the handoff lock into a network lock.

A retained test used 270 replay events, more than the live queue's 256-batch capacity. A truncation mutation initially produced a timeout, which did not establish the cause. After the test preserved terminal status, the failure became specific: 249 facts arrived instead of 270. The useful number is the missing content, not merely the fact that a test went red.

The writer also owns WebSocket control replies. A Ping requires a Pong control frame echoing its payload; an application JSON message named `pong` is a different thing. Network sends and outgoing trace writes stay with the writer, incoming traces with the reader. Neither belongs in the observation callback.

Disconnect uses a separate lifetime signal. Closing a data channel that a broadcaster can still reach is unsafe even if the broadcaster uses nonblocking sends. Shutdown joins writers before closing their logs. These are modest connection-lifetime rules, but violating them would let an optional receiver crash or delay the work it was meant to observe.

The optional transport adds `gui.log` alongside the event, API, and diagnostic logs. Each line contains an RFC3339 timestamp, a direction marker, and one compact JSON payload; the entire line is not itself a JSON object. Both directions matter. Seeing the server's output without the browser's command leaves

a different debugging hole from seeing the command without what the server sent back. General diagnostics still belong to Ensemble's logger; this is the transport's account of its own wire.

One Artifact, Several Ways to Lie

`ArtifactScroll` manages a list of Artifacts: thinking, chat, tool arguments, tool results, and user messages. An Artifact can accumulate content and then finalize; user messages and tool results need no invented streaming phase. Renderers handle markdown, JSON, or ANSI as appropriate, and CSS supplies differences such as dim thinking or compact tools. The component owns identity, updating, and finalization once rather than teaching a separate panel each version of those rules.

A delta finds or creates the part's card and appends through its renderer. A final replaces the provisional view with the authoritative rendering. Dispatch and finish observations give tool cards their execution state. The abstraction is useful because it shares a lifecycle, not because every artifact contains the same kind of text.

The first-edition implementation broke two basic display rules until its later audit. It built card content by interpolating values into `innerHTML`, and it capped displayed values without keeping the rest reachable.

A tool's name or arguments can contain filenames, URLs, and file contents influenced by outside input. Interpolating those strings as HTML moves untrusted markup into the page. The card shell must use `textContent` for such values rather than treating them as markup. Choosing a markdown renderer does not grant permission to assemble the surrounding card from raw

HTML interpolation. The security boundary reaches the human through the renderer, without requiring the model to obey a malicious instruction first.

Similarly, a shortened display must leave the full value available through a title or expander. An ellipsis without an affordance is not compact presentation; it has made the omitted content inaccessible. That becomes especially consequential when the hidden portion is the error needed to understand a tool result.

A later live Gemini page exposed a subtler failure. A legitimate empty text part displayed its own metadata. The renderer tested whether the text was truthy, then fell back to JSON. The empty string lost, and the card filled the vacancy with information that was never intended as prose.

The same fallback would expose opaque signatures and redacted thinking. Because automatic speech and the speaker button also used the card's text, the error travelled from the screen into both speech paths. Correct execution underneath did not make the page correct.

The repair is an explicit projection of readable content. Empty text stays empty. Readable Anthropic thinking and Gemini thought text can become prose; signatures and redacted bodies remain opaque replay material. Their bytes belong in History without being promoted into something to display or read aloud.

The source records a focused browser check using the retained Gemini part plus readable and opaque controls from an external peer. It examined the card, automatic-speech input, and speaker-button input separately. Restoring the fallback broke all three paths for empty or opaque cases while readable controls still produced native speech events. The saved part was enough to reproduce the bug. Another paid generation would have added randomness, not understanding.

Speech Is Also Flow Control

Automatic speech reads thinking and chat in full, abbreviates tool dispatches, and leaves tool results silent. The speaker on each card has a different purpose: it reads the full accessible text on demand. The Edition 3 page once displayed a long tool result but spoke only its short generated caption. Reusing the automatic summary for the speaker button had erased the distinction between a notification and a request to hear the content.

Network chunks are not phrases. The speech queue accumulates unfinished phrases by artifact identity, queues speakable portions, and flushes the remainder on finalization. A final must not read the entire answer again. Replayed history remains silent until the human requests it.

Speech also controls the tool gate. A tab's local reason to pause is queued or active speech, or user typing. Pause starts when an utterance is queued, not when audio finally begins; otherwise a tool can start while the browser is still waiting to speak its explanation. When the queue empties, the tab can unpause unless input still contains text. Sending the input reevaluates that condition. Escape with an empty input cancels speech and reevaluates it too. Typing must work as a pause cause even with automatic speech switched off.

This gives the speech queue an operational responsibility. If it gets stuck, tool starts can remain paused indefinitely. The source's Chrome behavior makes `onerror` as important as `onend`: cancellation can produce an `interrupted` error as ordinary queue behavior. Both callbacks must finish the current utterance and advance. The page keeps a reference to the active utterance and ignores an old cancellation callback after a newer utterance has taken its place.

The original Chrome work also found the speech service asleep after tab switches. Its recorded workaround is a zero-volume empty utterance on `visibilitychange`. The accepted page keeps that wakeup outside the audible queue, so a compatibility nudge does not acquire the pause gate. This is a recorded browser behavior and workaround, not a claim that every browser version needs the same treatment.

Multiple tabs introduce a more interesting policy than simply OR-ing every browser's desires forever. Pause is Agent-wide. An explicit **Unpause everyone** releases the one gate even if another tab still contains unsent text or queued speech. That other tab's unchanged local cause cannot veto the decision.

Automatic pause commands therefore follow changes in the combined local cause, not every utterance boundary. Repeatedly asserting pause while the cause remains true would undo another tab's explicit unpause. Reconnection must not reassert stale local pause state either. The server owns effective Agent state; each tab contributes transitions, not a permanent lease on the gate.

The policy trades unanimous consent for a usable override. Another tab's genuine new transition can affect the gate, but its old condition cannot keep winning by repetition. That distinction matters more than whether the button looks reassuring.

What the Browser Evidence Establishes

A WebSocket test can establish message fields and replay content. It cannot establish safe DOM construction, the text supplied to speech, or whether cancellation advances a native speech queue. The source keeps those checks separate rather than treating transport success as proof of a usable interface.

Its retained browser work covered two tabs, connection during an active partial, completed-history reload, safe markup, reachable long content, typing with speech disabled, cross-tab unpause, speaker clicks, Escape, and visibility changes. The historical browser receipt records native speech events. Those establish API activity, not that a person heard intelligible audio.

The speech work did not pass cleanly at first. One multi-tab record included a native `canceled` error. A later repeat waited for speech that never completed: reloading the second tab had restored its checked Read aloud default and accidentally enabled overlapping speech during the long response. Explicitly disabling voice in that tab after reload produced the clean automatic-speech result. Reloading had changed the experiment's conditions, not merely its appearance.

A separate interruption record exercised a real `speechSynthesis.cancel()`: the first utterance started, cancellation occurred, and the second started and ended with pause released. Restoring the defective error handler stranded the queue. Listeners had to be attached before playback so a fast start could not escape observation. This demonstrates queue recovery in the native browser; listening remains a separate judgment.

The source's live summary records historical Anthropic, OpenAI, and required Gemini 3.8 turns that ran `run_command` and returned their markers successfully. The Anthropic run also delivered a terminal busy hint to the browser on the same Agent. Gemini's successful tool run was the one that revealed the empty-text metadata card. The later projection correction repairs that display defect; successful tool execution had never proved the rendering right.

These are retained source results, not new executions for this companion or a current provider-availability list. Their value is the distinction they expose: a system can finish the requested

work, show the wrong thing, and leave the human unable to steer the next action.

The chapter leaves one scroll pane and one input field, with a reusable unit for both streaming and completed content. The next chapter arranges those units into a workbench with separate tool views, an agent tree, and workspace controls. The layout can grow because the harder agreements are already in place: the Agent owns the work, History owns the facts, and a browser can come and go without becoming either.

Chapter 8: Build Your Dream GUI

The settings server saved a speech rate of 1.75. After a restart, the label still said 1.75. The native speech API received approximately 1.8.

In the source chapter's account, an independent reviewer followed the value past the successful WebSocket echo. The browser's range input had a step of 0.1, so Chrome rounded the saved value to a legal slider position. The label read the settings object. The speech queue read the slider. Each had a plausible source of truth, and the person using the application got the difference.

Chapter 7 established a view from which a human can watch and steer one Agent. This chapter makes that view somewhere a human might choose to work: separate chat and action panes, a sidebar, adjustable widths, coherent themes, and preferences that survive a process restart. The additional server machinery is small. The judgment lies in deciding which state belongs where, when a change takes effect, and what evidence shows that it reached its destination.

For Bill, speech is central to using the agent, not an ornamental accessibility checkbox. The broader goal is to start doing useful coding inside the new system early enough that daily use exposes its missing features.

Two Views of the Same Work

The workbench has a left sidebar, chat in the center, and actions on the right. Two dividers control the widths. This is a change in presentation, not a new execution architecture.

Chapter 7's `ArtifactScroll` now has two instances. User messages, readable thinking, and reply text go to Chat. Tool-call argument streams and their final parts go to Actions, followed by dispatch and finish observations and tool output. Status messages update workbench controls rather than becoming transcript content.

The routing must recognize both a part's kind and the messages that later update it. A tool's arguments can arrive before its dispatch observation. Preserving artifact and call identity lets dispatch update the card already showing those arguments; it must not create a second, apparently unrelated execution. The same rule applies to replay. Moving content to a different pane does not change what event it represents.

Both panes retain the same streaming protocol, renderer lifecycle, and styling vocabulary. Routing is a filter over one component, not permission to build a second tool renderer.

Each pane scrolls independently. Following a long tool result should not drag the human away from a reply being read in Chat, and the follow control needs its own space rather than covering transcript text.

The dividers are similarly modest. A drag changes a width, with a minimum preventing a pane from collapsing into uselessness. The accepted page described by the source uses pointer capture so a gesture continues after the pointer leaves the narrow divider. Releasing the gesture ends its tracking. Focused dividers also accept arrow keys and expose their current value through separator attributes.

There is no prescribed visual style. There is a requirement for judgment: strong contrast, readable text, clear grouping, and tabs whose selected state is unmistakable. A row of generic buttons may switch panels correctly while failing to communicate that the panels belong to one workspace. Browser measurements can establish geometry and control behavior. They cannot decide whether this is a place Bill wants to spend the afternoon.

The Settings File Is Not the Agent

The sidebar switches between Agents and Settings. Four visible controls cover theme, automatic speech, speech speed, and font size. They send changes through the existing WebSocket. No model call is necessary to remember a font size.

The source records an ownership correction here. Browser settings and their file handling were initially placed in the core Agent, then moved back into the optional GUI module. The Agent owns its configuration, but that does not make every configurable thing an Agent concern. A theme does not become execution policy by sharing a JSON object with it.

The GUI server owns the browser preferences, their mutex, and their persistence path. The application chooses that path; the reference command uses `cr/settings.json` beneath its working directory. Provider credentials remain in the intended process environment, not in this preferences file, logs, commands, or browser controls. There is no API-key or model-selection panel in this chapter.

One field crosses the boundary deliberately: `max_tool_rounds`. It already belongs to the Agent's execution policy and travels in the saved settings envelope so a restart can restore it through the public Agent interface. It does not need a

fifth visible control. Settings snapshots obtain that policy from its Agent owner rather than allowing the GUI to maintain a competing runtime value.

The distinction matters during failure diagnosis. The optional server can remember and display the requested budget without the actor ever using it. Conversely, an execution policy can work perfectly while the browser falsely reports a different value. Storage, display, and execution are related responsibilities, not interchangeable proof.

Restart has a narrow meaning here, too. Reusing the working directory restores preferences and the next-turn budget. It does not restore a saved conversation. Chapter 7's browser reconnect recovers a view of a still-running Agent; this chapter's settings load survives a new process. Neither supplies general Agent save/load.

A Patch Comes In; a Snapshot Goes Out

Consider a tab turning Read aloud off. Its `update_settings` message contains `tts_enabled: false`; it need not resend the theme, font, speed, or budget. The server interprets this as a patch over the accepted settings. After a successful update, `settings_changed` carries a complete snapshot to every connected client, including the sender. A new subscriber receives `current_settings` alongside the existing history and active-part handshake.

Absence means different things in those two directions. In a patch, an absent field means "leave it alone." An explicit false means "turn it off." An explicit zero can select a policy default. In a snapshot, those values are part of the answer and must remain present.

An ordinary Go boolean with `omitempty` cannot represent both an absent patch field and a request to set false. The small adequate design decodes a sparse object over a copy of the current accepted value, or tracks presence explicitly. The full snapshot omits neither false nor zero. Otherwise, the server can accept a change and then send a reply incapable of describing it. A client may keep showing the old value, or even a number the server has just clamped away.

The copy is also an error boundary. JSON decoding can encounter a type error after reading earlier fields. Decoding directly into live state would let a rejected message leave half its changes behind. Decoding into a candidate keeps the accepted state intact until the whole candidate is valid. Input must be a JSON object, not merely something the JSON decoder can consume.

Validation occurs on entry, for both wire updates and disk loads. The same path checks the theme vocabulary and clamps the bounded preferences. In the reference design, an unrecognized theme falls back to dark. The particular font-size endpoints are implementation choices, not a new core policy. A missing settings file starts from defaults; malformed saved JSON produces a startup error instead of silently discarding the person's preferences.

The source's first-edition audit found a temperature of −5 reaching the engine because both update and load paths trusted their input. That is a historical lesson about validation boundaries, not a reason to add a temperature control here. Checking only at the point of use would leave invalid accepted state available to every other consumer. Checking only network input would let disk restoration bypass the repair.

The budget has two boundaries with different duties. The public Agent interface rejects negative budgets. The GUI's shared validator may normalize a negative saved or incoming value to

zero before calling that interface. The GUI accommodates its input; core protects its invariant. Teaching the Engine to repair browser preferences would muddle both responsibilities.

Save Before Announcing Success

A valid candidate is not yet an accepted update. It still has to be saved.

The original settings sketch ignored decoding and write errors while mutating live state. That creates an attractive failure mode: the current tab works, other tabs agree, and a restart discovers that none of them was describing durable preferences.

The accepted source design writes a small replacement file and renames it over the destination before publishing the candidate. If writing or renaming fails, the previous accepted value stays active and no success snapshot goes out. The server reports the failure, and the page displays the existing `settings_error` reply. After a successful save, the server restores the budget through the public Agent setter and publishes the complete accepted settings.

The existing server mutex orders application, persistence, publication, and new subscriptions. A client subscribing during an update should see a coherent accepted value, not a mixture of the old snapshot and the new policy. This is coordination around one small file, not a transaction framework. Replacement by rename is not a claim of power-loss durability.

Using the existing WebSocket keeps settings on the same ordered connection as observations and subscription state. Adding a REST settings path would require reconciling an independently timed fetch or update with that stream, especially after a tab sleeps. The source avoids that extra reconciliation rather than declaring that two transports can never work.

This does not upgrade the socket into a reliable-message service. Chapter 7's slow-client progress limits remain. Broadcasting a successful change to connected clients and supplying the accepted snapshot on subscription are the intended synchronization paths; they are not a promise that every overloaded tab sees every intermediate update.

A Budget Belongs to a Turn

The source records that `max_tool_rounds` once persisted correctly while two orchestration loops ignored it and used a hard-coded limit of 16. The GUI could faithfully save 200 while execution stopped at sixteen. This defect survived longer than the early settings audit because all the storage behavior looked healthy.

Chapter 5's design now gives the policy one owner. Zero selects the named default of 200; positive N permits N tool batches per human turn. The unit is a batch, not an individual tool call, a provider request, or the total number of processes still alive. The Agent captures the setting at turn start.

Capturing it matters because a settings update is not an interruption command. Suppose a turn starts with a two-batch budget and another tab raises the saved budget while that turn is running. The settings snapshot can immediately report the newly accepted setting, but the active turn still has its original allowance. The next turn receives the new one. Reading live configuration at every dispatch would let a different tab quietly rewrite the bounds of work already under way.

At the captured limit, the actor retains Chapter 5's final-response allowance. A stop-tools instruction goes into request-local prompt context so the model can finish its answer. It does

not alter History, modify signed content, or leak into later turns. Remembering a budget must not corrupt the conversation used to enforce it.

The source describes an entry-point test that connects these pieces rather than inspecting the saved JSON alone. It saved a two-batch limit and restarted the actual two-view application. While the first provider response was held, a settings update raised the saved limit to eighteen. The active turn stopped after two batches; the next terminal turn completed eighteen. A further restart retained eighteen, and zero allowed execution beyond the old sixteen-batch ceiling by selecting the default of 200.

Only the external provider was fake in that test. The actor, tools, terminal input, WebSocket, and settings file remained real. Restoring the live-read defect made the active turn execute eighteen batches instead of two. That is evidence about when the policy takes effect, not evidence of a live provider's capabilities.

Controls Must Describe Their State

Theming is mostly a useful exercise in keeping presentation centralized. Semantic CSS custom properties provide colors for surfaces, text, borders, controls, code, errors, and tool cards. Changing the active theme changes those values without reconstructing the transcript. Dark is the source's default because this workbench was built around Bill's speech-oriented use; that is a design choice, not a universal accessibility rule.

A saved `system` preference is different from a saved dark or light choice. The page resolves `prefers-color-scheme` to a concrete theme and listens for changes while system mode remains selected. A one-time lookup follows the operating system only until it changes. The saved preference remains system even though the displayed colors are currently dark or light.

Control state also needs a semantic representation. A CSS class can make a toggle look selected while telling assistive technology and automated observers nothing about its state. The relevant control attribute must say what is selected, enabled, or pressed; CSS then styles that state. The same distinction applies to sidebar tabs and dividers.

The agent tree supplies a small but memorable example of false agreement. In the source's implementation history, both the CSS and an initial browser check used state names copied from an example: `thinking` and `tool_use`. The actual reducer emitted `in_flight` and `tools_pending`. The stylesheet and the test agreed with each other. The active indicator did not pulse.

The corrected check inspected the active node's computed animation using actual Agent state. Restoring the old selectors broke it. An attribute that a test wrote itself was not evidence that production events reached the rendering rule.

The sidebar's data model is a tree, and its renderer walks children recursively, but there is currently one node. Its indicator becomes active for working states and dims when idle. An empty children array leaves room for later display growth; it creates no child Agents, assigns no ownership, and establishes no sub-agent lifetime or protocol.

Speech Has Several Consumers

Settings application handles both the initial `current_settings` and later `settings_changed` snapshots. It updates visible controls and the behavior behind them. Updating only the change path would make a newly opened tab display defaults until someone touched a setting.

Read aloud controls automatic speech. Turning it off cancels that speech, but explicit Speak requests still play. A changed rate applies to the next utterance, not retroactively to sound already under way. These distinctions preserve the difference Chapter 7 established between automatic notification and a human asking to hear a specific artifact.

The interaction with typing is more subtle. Changing Read aloud releases the old typing pause without deleting the draft. Fresh typing must still veto automatic continuation, even if an acknowledgement of an earlier change arrives afterward. Explicit Continue in any tab releases that veto. An implementation that merely clears pause whenever a settings echo arrives can erase a newer human action with an older reply. The text survives, yet the boundary the human expected to hold has opened.

The fractional-rate failure at the start of this chapter lives at a different boundary: the saved preference entering a native control. The correction was to let the slider preserve arbitrary fractions with `step="any"`, with a comment explaining why tenths were insufficient. Changing the label would not have helped; the label was already right.

The source's focused check saved 1.75, restarted the real application, inspected the snapshot, label, and slider, then submitted browser input and recorded the native utterance's rate. It repeated the sequence with 1.2345 so a smaller fixed step could not merely relocate the defect. The historical baseline and restored-defect record are linked by the source. With the old step restored, the two requested values became approximately 1.8 and 1.2 at the native API. Both correct and defective utterances still produced start and end events. Completion was never proof of the requested rate.

A separate failed check confused floating-point representation with a product defect. A saved 0.6 arrived at the native API as `0.6000000238418579`, so exact equality timed out. The

focused regression used a tolerance of 0.00001: enough for that representation noise, far too small to hide the rounding from 1.75 to 1.8. The source also retains an earlier timeout without enough location detail to identify its cause. It does not recruit that failure as additional evidence for a diagnosis it cannot support.

There is one further distinction between the historical demonstration and the current source design. The fractional-rate check used the earlier control range, which topped out at 2. The later crossover extended the saved preference and control to 5, while every native utterance construction clamps the rate to 0.5–3. Thus a saved 2.5 reaches native speech as 2.5; a saved 5 reaches it as 3. The old test establishes fractional preservation within its tested range, not verification of that later extension.

Preserving fractions in the control and clamping at the native consumer solve different problems. The first prevents accidental quantization of accepted input. The second deliberately maps the application's approximate preference into its chosen native range. That mapping must cover both automatic speech and explicit Speak. Native voice rates are nonlinear, so a value labelled 5 is not a measured fivefold speedup.

Snapshots establish what was accepted. Slider values establish what the browser control retained. Utterance rates establish what the native API received. Native callbacks establish API activity. None establishes that a person listened and found the result intelligible. For a speech-centered workbench, that final judgment is not an optional embellishment.

Usable Enough to Expose the Next Problem

The source retains browser work covering two-tab settings changes, system-theme changes, pointer and keyboard dividers, pane routing, the active Agent indicator, and visible restoration after restart. Those historical checks used the earlier speech control range. The separate fractional-rate investigation matters precisely because a tenths-aligned value in the broader browser run did not expose rounding.

It also records a historical live OpenAI turn: saved controls restored after restart, real shell output appeared in Actions, the answer remained in Chat, and a terminal hint reached the same Agent. The answer's speaker produced native callbacks. Unchanged provider coverage was reused from Chapter 7 instead of paying for a fresh matrix merely because settings had changed. These are results from a specific historical run, not present provider-availability claims.

Bill's stated destination in the source is to run GPT-6 Astra at high reasoning in the new Ensemble and use that agent, rather than Codex, to build Edition 3. That is a target, not a report that access, reasoning configuration, or the coding crossover has been verified. A usable workbench brings the target closer by making ordinary work possible; it does not certify that the switch has happened.

Chapter 9: Skills

Chapter 8 follows a preference from its saved value to the behavior it actually controls. Skills pose a similar question at a larger scale: when an Agent says it has learned a capability, what has changed?

A new paragraph in the conversation is not necessarily a new instruction. A newly advertised tool is not necessarily executable. A hidden tool may still execute if dispatch never checks whether it is enabled. And a successful call after unloading may prove nothing about the capability supposedly being unloaded.

The source chapter contains that last trap in miniature. A historical OpenAI session loaded `code-tools`, wrote a scratch file, marked the skill for unload, and then successfully read the file. It looked like evidence that pending unload preserved the acquired tools. But `read_file` had belonged to the primary skill all along. The demonstration had carefully tested the lifetime of the wrong tool.

Skills make these distinctions explicit. Each skill packages instructions, tools, and dependencies in a named Markdown file. The Agent can acquire a new manual and the means to act on it without becoming a different Agent. The interesting work is keeping those two halves together, through dependency resolution, prompt rendering, provider protocols, and execution.

A Small File, Several Different Powers

The workbench now has one Agent worth using. Adding every future capability directly to its system prompt would turn that prompt into a communal cupboard: identity, operating instructions, project facts, tool descriptions, and preferences all squeezed together. Any edit could disturb the supposedly stable prefix, and the Agent would receive material unrelated to its current task.

A skill supplies a smaller unit of composition. Its file lives at `skills/<directory>/SKILL.md`, with frontmatter followed by an ordinary Markdown body. The frontmatter gives it a name, a description, a type, and lists of tools, dependencies, and further skills it offers. The body supplies its instructions. This is packaging for existing capabilities, not a way to manufacture handlers by naming them. The source corrected an earlier example that advertised an undefined `think` tool. A name in a manual cannot implement a function.

The three skill types have different jobs. A `primary` skill defines the Agent's initial identity and loads at creation. A `loadable` skill can be offered for later acquisition. A `dependency` loads automatically when needed and stays out of the Agent's discovery list. Both the primary and its startup dependencies contribute instructions and tools. When skills are configured, `load_skill` and `unload_skill` remain available so the Agent can manage acquisition even if the primary declares only a reader.

The file format is deliberately smaller than general YAML. Scalar fields and two list forms suffice: space-separated names on one line or an indented list. The parser requires an opening frontmatter boundary, a name, and a valid type, and reports malformed files with their paths. It splits only at the frontmatter

boundaries, leaving later Markdown horizontal rules in the body. Otherwise, an innocent separator halfway through a manual could become a configuration event.

The directory locates a definition; the declared name identifies it in dependency lists and load requests. Duplicate names are errors. Letting directory traversal order choose which definition wins would make the Agent's identity depend on an accident of discovery.

None of this discovery grants capability. Finding a file means the registry knows a definition exists. Offering it means the Agent may ask to load it. Loading it means its instructions and tools become active. Those are separate states, even when a small example makes them happen close together.

A Load Must Succeed as a Whole

Consider the source's small scratch-file example. The primary `base` skill enables `read_file` and offers `code-tools`. Loading `code-tools` adds `write_file` and requires a hidden `helper` dependency, which adds `list_directory` and instructions to stay in the scratch directory. The next model request must carry both manuals and the newly enabled tools. The dependency's instructions are not optional decoration attached to its function names.

The registry first resolves the entire dependency graph. A visiting set detects a cycle such as A depending on B depending on A. A resolved order lets dependencies activate before the skill that needs them. An unavailable requested name, wrong type, missing dependency, or cycle rejects the load without partially activating the graph.

The order matters more than the recursion. Suppose the first dependency grants a writer and the second is missing. Enabling as traversal proceeds would return an error while leaving the writer available. The caller asked for one capability and received a failed request with a useful side effect. Resolving first and activating afterward keeps failure intelligible.

After activation, the available-skills list is recomputed from the `loadable_skills` lists of all loaded skills. Only discovered definitions of type `loadable` qualify, and already loaded skills disappear from that list. A dependency does not become directly loadable because another file happens to offer its name.

This produces progressive disclosure. `code_tools` can offer `refactor_tools`, which can offer something more specialized. The Agent sees the next choices made available by its current manuals, rather than the whole directory as a menu. That does not require the definitions to form a literal tree: dependencies and shared offers can make a graph. The useful boundary is reachability from what has already loaded.

Progressive disclosure reduces what the Agent must consider, but it is not itself an execution guard. The enabled-tool set must govern both provider declarations and local dispatch. If a hidden `write_file` handler remains callable through the installed-handler table, the model can bypass disclosure simply by producing the name. The ordinary response to that guessed call is a tool error, with no write. After the skill loads, the same installed handler can execute.

This boundary controls admission to the tool. It does not turn the helper's scratch-directory prose into a filesystem sandbox. Instructions describe how the Agent should use a capability; dispatch determines whether that capability is available at all.

One Agent Owns the Registry

Several Agents can read the same skills directory without sharing a mutable skill state. Loading a writer into one must not grant it to another. The primary choice, enabled tools, offered skills, rendered bodies, and pending removals belong to the particular Agent.

That ownership does not justify importing the skills implementation directly into the Agent package. The existing star-shaped package structure still applies. Common holds the lifecycle interface and shared types; the skills package owns parsing, traversal, and rendering; the composition root constructs the registry and injects it into the Agent. The registry also receives its required Agent parent through the common interface. Both sides are attached before the primary loads, because rendering can consult the owning Agent's declarations.

The application supplies construction inputs. The CLI and optional GUI translate `EN_SKILLS_DIR` and `EN_PRIMARY_SKILL` into a per-Agent skill specification. The registry does not discover its identity by consulting process environment whenever it needs an answer. An embedding application can therefore create different Agents in one process without a process-wide primary rewriting their identities.

Application variables follow the same boundary. The command's `EN_CUSTOM_VAR` is one example of translating configuration into a rendering callback, not a reason to give the registry environmental authority. Omitting the optional skill specification preserves the earlier public tool-registration behavior. Skills are a way to assemble an Agent, not a forced migration for every existing consumer.

Render Once, at the Relevant Time

The primary and other startup bodies form the initial system prompt. Their text is rendered at creation and remains intact. Later loads render new bodies and add them to that instruction context. The initial identity survives; the collection of manuals grows.

`$TOOLS` expands from the owning Agent's currently enabled declarations. `$SKILLS` expands from its currently loadable set. Application variables use registered callbacks. Constructor-supplied variables must be installed before primary rendering, or the first and most consequential body would miss them.

A dynamic body renders after activation, so it can describe the tools just acquired and the next skills just revealed. The original primary's `$SKILLS` expansion does not get rewritten to match. It records the choices at creation; the new manual supplies the later view. This is an intentional sequence of rendered instructions, not a live template silently rewriting every earlier paragraph.

The same distinction applies to application context. If `$CUSTOM_VAR` renders as a creation-time value in the primary, replacing its callback changes future renderings, not that existing text. In the accepted implementation described by the source, unknown placeholders remain verbatim. A visible `$CUSTOM_VAR` is easier to diagnose than an unexplained hole in a sentence.

Rendering callbacks run outside the registry lock. This detail is easy to lose in an otherwise tidy design: `$TOOLS` asks for declarations, and declarations consult the same enabled-tool state. Calling the callback while holding that lock can deadlock the first manual useful enough to mention its own tools.

Substitution also moves data across a privacy boundary. Whatever a callback returns becomes model-visible instruction text. A convenient environment variable is not automatically an appropriate variable to render, particularly when it contains a credential.

Skills do not make every piece of context an instruction. Memories, project notes, and changing user-preference data belong in message history as data, separate from the system's operating instructions. This chapter adds neither memory storage nor a memory-saving tool. Keeping that distinction now avoids building a constitution out of yesterday's project notes.

Instructions and Tool Disclosure Travel Separately

The simplest delivery path puts all startup instructions in the system prompt and startup tools in the provider's normal tool declarations. A later load adds rendered instructions to the system prompt and updates the ordinary declarations. That is the source chapter's policy for providers other than Anthropic, and the fallback for Anthropic cases that cannot use incremental disclosure.

It costs prefix stability. Changing SYSTEM can invalidate a cache, and changing tool declarations can do so too. Preserving an old prefix by putting new operating instructions only in a tool result would preserve the wrong thing. A tool result can report that loading succeeded; it is not a substitute for delivering the loaded manual in the provider's instruction context. The current source explicitly supersedes its earlier frozen-system design on this point.

Anthropic's accepted path in the source separates tool discovery from instruction delivery more finely. Under the documented tool-search protocol used there, the full tool catalog still travels in every request. Startup tools use ordinary declarations. Undisclosed tools carry `defer_loading`. A successful model-called `load_skill` reveals newly acquired tools through `tool_reference` blocks in that call's paired `tool_result`.

The wire result's shape matters. The source records an accepted Edition 3 exercise in which mixing human-readable load status with `tool_reference` blocks caused HTTP 400. A corrected result containing references only was accepted, and the Agent then used the loaded writer. The helpful sentence was the protocol error.

Canonical history still records the textual load status. The provider renderer projects the successful discovery result into reference-only content for that provider. This preserves the application's readable history without pretending that its canonical representation is always the wire representation.

The references disclose tools, not instructions. The documented API used by the source supplies no separate mid-turn instruction channel, so loaded manuals still join SYSTEM. Keeping the catalog stable can avoid unnecessary declaration changes; it cannot guarantee a cache hit when the instruction text grows.

There are limits to when this form is legitimate. A direct or manual load has no model call whose result can carry the references. A load without a paired discovery result has the same problem, and an unsupported model cannot use the protocol merely because the application prefers it. These paths use ordinary enabled tool definitions. The renderer must not invent a model call or enabling event to make the incremental representation fit.

These are the provider mechanisms and observed correction recorded by the current source, not a fresh claim about every model's present support. The distinction is especially useful here because the source also preserves earlier live Claude, OpenAI, and Gemini sessions. Those sessions loaded the example graph, wrote and read their markers, marked a skill pending unload, and exited normally. Their captured system text stayed identical within each session. They establish behavior of the earlier frozen-system design, not verification of the instruction delivery now required.

Provenance Makes the Projection Possible

The provider renderer needs to know whether a tool belongs in the startup declarations or was enabled by a particular discovery call. It does not need to know how a skills directory is organized.

Each enabled tool therefore records its first enabling source and event sequence. Initial tools have sequence zero. A dynamically acquired tool records the existing `tool_called` sequence of the actual `load_skill` call that enabled it. There is no separate skills counter to reconcile with conversation history.

First means first. If the primary already supplies `read_file`, another skill mentioning it does not reclassify the reader as newly acquired. It retains its initial provenance. Otherwise, the renderer could try to disclose an old tool through a new event, or later lifetime reasoning could mistake a shared capability for one introduced by the latest skill.

This metadata lets the Anthropic renderer pair references with the successful result that actually enabled them, without importing the registry implementation. It also distinguishes a

manual load, which has no enabling model call and uses ordinary declarations. A stable event identity connects the application's capability transition to the provider's conversation structure.

Pending Unload Means Still Loaded

Loading has an effective transition now. Unloading does not yet have its removal boundary.

`unload_skill` accepts a loaded name, marks it pending, and reports that state. An unloaded name is an error. Pending unload retains both instructions and tools. It does not rewrite prior manuals or disable their handlers. Actual removal is deferred until the later compaction boundary; this chapter records the intent without implementing that compactor.

That is not merely reluctance to delete a map entry. The model has already received instructions and, potentially, disclosed tool definitions. The application needs a coherent point at which its capability set and the model's context can agree about removal. Anthropic's context-editing facility can clear selected tool results, but that does not semantically retract a manual or revoke an application capability. Deleting one piece of conversation content is not a general unlearning operation.

The historical unload demonstration at the start now has a precise diagnosis. Reading after pending unload exercised a startup capability, so it could not reveal premature removal of the dynamic writer. The source's focused test instead checked that `write_file` remained declared after pending unload. Declaration and dispatch use the same admission predicate, and restoring early removal made that check fail. The revised grader also exercised a dynamic write after marking unload. Those checks concern the capability whose lifetime is actually changing.

The Agent can now expand its working instructions and executable tool set without replacing its initial identity. Its registry has one owner, loads are whole-graph transitions, provider projections have real event anchors, and pending unload says exactly what has not happened yet. None of this saves the conversation across process death. The next chapter takes up that separate lifetime: allowing the conversation to outlive the process that created it.

Chapter 10: Persistence

Chapter 9 gives an Agent new instructions and tools without replacing its identity. None of that helps when the process exits. A useful afternoon of work can still end with the next launch meeting a stranger.

The first persistence design made remembering optional: it had a `--load` flag. Bill's review, quoted in the source chapter, was one line: "Let's load by default without a flag."

That changes more than the command line. If loading is an explicit experiment, a bad file is an experiment that failed. If loading is the normal start of work, a bad file is the user's conversation in trouble. Starting fresh and saving cheerfully on exit can destroy the very thing the user returned to recover.

The design in this chapter makes saving and loading ordinary lifecycle operations. It also puts the earlier event architecture under a useful strain. The Agent has a snapshot of what the conversation means and a history of how it got there. Across a restart, those two accounts must agree.

What Actually Survives

The save is one JSON object. Its recovery vocabulary is small: `context` holds the snapshot, `as_of` identifies its event boundary, and `log` holds the available canonical events. `config` records the request configuration at the time of saving.

Optional `log_mode` and `dialogue_log` distinguish a full archive from a derived dialogue archive; older four-field saves mean full mode.

The context is what the renderer reads: dialogue, its structured parts, and the other state already defined by the event model. The log is the sequence of facts from which the reducer derives that state. A snapshot is not a second interpretation of the conversation invented for persistence. It is the renderer's existing context, saved.

That distinction limits what "the Agent remembers" means. A saved tool declaration does not become a handler closure on restart. A recorded job does not resume as a running process. The save does not reconstruct the skills registry's loaded or pending-unload state. The new Agent supplies its current tools and primary skill, while the restored conversation can still contain previous tool results and manuals. Persistence carries conversation state, not the whole machine that once produced it.

The recorded configuration contains vendor, model, effective system prompt, and tool declarations. API keys and base URLs are excluded from that metadata. More importantly, the metadata does not configure the new Agent. Today's model, prompt, tools, and settings win. Pinning restoration to yesterday's model would make a model retirement a conversation-retirement service.

The source records a quieter mistake in the earlier Edition 2 implementation. Its first save code copied the configured system prompt. For an ordinary Agent without a skill, that field was empty, but Engine supplied a nonempty default to the actual request. The conversation resumed correctly; its metadata described instructions the model had never received. A test using a primary skill missed the discrepancy because that path had already filled in the configured text.

The correction asks Engine for the effective instruction through its existing owner interface, sharing the decision with all three renderers. Even that has an exception worth keeping: an intentionally empty primary skill must stay empty. Replacing every empty string with a default would repair the no-skill case by breaking the empty-skill case. The source's focused checks compared saved instructions with the captured HTTP field and checked the empty primary separately. The useful lesson is not to normalize more aggressively. It is to ask the component that owns the decision.

The Anchor Makes Two Copies Useful

Suppose a snapshot contains everything through event 7, while the available log runs through event 10. The loader installs the snapshot, then applies 8, 9, and 10. It neither replays the first seven events nor ignores the final three.

`as_of` makes that possible. It is the sequence number of the last event folded into the snapshot, not the number of retained log entries. Without it, a loader has no principled way to choose between applying everything twice and omitting everything newer than the snapshot.

With a non-null context, restoration applies only canonical events whose sequence is strictly greater than the anchor. With a null context, restoration begins from an empty context and applies a complete canonical history. The reducer is the same in both paths. Snapshot-only recovery is also legitimate: a non-null context and an empty log contain everything the renderer needs to continue.

A normal full-history save usually has an anchor equal to its last logged sequence, so it has no recovery tail. This makes a broken loader surprisingly easy to admire. A loader that always

ignores the tail can pass round trips using every file its own save routine produces. The revealing case pairs an older snapshot with a newer log. The source deliberately tests that combination. It exercises the shape needed for later crash recovery without claiming that this chapter already writes events durably during execution. Here, saving happens at normal exit; Chapter 14 adds journaling.

The snapshot and anchor must also be captured together. Copy the context just before an event arrives and the anchor just afterward, and the save claims to contain a fact it has never seen. History already owns both values, so its existing lock can protect the pair. A new persistence coordinator would add another owner to a problem that already has one.

Restoration has a similar ownership boundary. Decode into a fresh candidate, validate the configuration, snapshot state, anchor, events, and strictly increasing sequence numbers, and stage the resulting context and log before installing them. Even an event below the snapshot boundary must be valid, although it will not be applied again. A malformed late event must not leave a live Agent with the first half of a restored conversation.

Validation cannot simply demand the tidiest possible transcript. The public `Tools().Dispatch` path can produce tool events outside a model turn without a native call ID. That absence is legitimate history and must remain absent. Tool calls inside a model turn still need their supplied correlation IDs. Inventing IDs to make an archive look uniform would alter the facts it is supposed to preserve.

Loading is local reconstruction. It does not ask a provider to explain the old conversation, nor should merely opening a restored GUI view make a provider request.

Two Roads to the Same Request

Persistence makes the Chapter 2 promise falsifiable: context is a pure function of events. No clock, unstable map traversal, or engine mutation behind the reducer's back may supply a fact needed to reconstruct it.

A process can violate that promise for its entire lifetime without noticing. Its live context is the only copy anyone consults. A save creates a second account. For a complete canonical history, loading the snapshot and rebuilding from the log should produce byte-identical vendor requests for the same next prompt under the same runtime configuration.

The request is a useful comparison boundary because it is what the model receives. It avoids requiring an outside observer to know the Agent's internal representation. It also catches failures that a plausible answer conceals. A model can remember a word despite duplicated turns, lost tool results, or missing opaque provider material. Correct recall is welcome; it is not proof of faithful reconstruction.

The source's historical Claude, OpenAI, and Gemini sessions each remembered `cobalt-juniper` in a second process without being given the word again. Claude also remembered it from independent snapshot-only and log-only variants. Those are reported restoration observations, not current provider coverage or proof of request equality. The source's deterministic checks did different work: they exercised all three renderers with tool identities, opaque material, a redaction, and pending ephemera, comparing snapshot, full replay, older-snapshot-plus-tail, and snapshot-only paths.

There are two kinds of equality here. Rendered requests must match byte for byte under the fixed comparison configuration. Comparing a reconstructed context with its saved snapshot should compare decoded values, not JSON indentation or

whitespace inside encoded tool arguments. Nil and empty dialogue or ephemera collections can be normalized for that comparison. Numbers must retain their exact values; rounding a large integer could hide a changed tool argument. Representation tolerance does not excuse missing fields, different facts, bad anchors, or disordered sequences.

The complete-history condition is not small print. Once a log prefix has been discarded, a later save cannot manufacture it. A snapshot-only save can resume correctly, acquire new events, and save again, but its resulting log is still not a complete account from the beginning. Nulling that snapshot is not a replay test. It is deleting the only surviving representation of earlier state. The loader must refuse a rebuild whose originals are unavailable.

The source also preserves a useful false alarm from the fresh Edition 3 work. Default-load checks failed while snapshot, tail, and refusal checks passed. The fixture supplied `CURRENT_RUNTIME_PROMPT` with a terminal newline; its exact expectation had none. Correcting the fixture input resolved the mismatch with production code unchanged. Trimming prompts in production would have made the test happy by changing a different behavior. Restoring conversation and selecting current runtime instructions remain separate operations, and a failing comparison is evidence to investigate, not permission to blur that boundary.

Identity Cannot Restart at One

An archive can lose log entries without losing sequence identity. If its snapshot is anchored at 7 and its log is empty, the next event is 8. If it has a tail ending at 10, the next is 11. Allocation continues one past the larger of the anchor and the last retained event sequence. The length of the array is irrelevant.

The browser has a related problem with different ownership. A provider tool-call ID correlates a call and result. A transcript card address identifies something the user can select. A parser's turn-local counter is not automatically a safe address for a card that survives process restart.

The source describes the fresh implementation having to preserve old card addresses while allocating new ones beyond the restored conversation. Otherwise, a new stream can append to an old card or an old control can select a new result. Remembering the text while moving its buttons to somebody else's result is only partial success.

Preventing future collisions is not enough when a legacy archive already contains colliding parser-local IDs. Those old artifacts still need distinct presentations, deterministically across replay and restart. The display must repair its addressing without rewriting provider payloads or call/result correlations. The archive records what happened; the GUI decides how to address its presentation. Neither needs a new public identity framework merely to respect the other's ownership.

This is also a boundary on evidence. Backend replay equality cannot establish browser selection behavior. The source reports separate DOM assertions against a local fake provider across a GUI restart. Those assertions checked old and new card identities; screenshots were not a human visual or audio review, and the fake provider was not a live-provider test.

Retain the Conversation, Not Every Diagnostic Forever

The model never reads the canonical log directly. It reads the context. Already-folded events can therefore be removed from a save without changing the next request, provided the snapshot

remains. What disappears is the ability to reconstruct or diagnose discarded history, not necessarily the ability to continue the conversation.

The source offers two saved-history modes. `full`, the default, retains all available canonical history. "Available" matters after a trimmed save: changing the setting cannot recover a missing prefix. `dialogue_only` keeps an ordered, derived archive of visible user and assistant text, with source sequence numbers for provenance. It excludes tool calls and results, opaque reasoning, private replay payloads, and API or GUI traces. For an assistant event containing several kinds of material, only its visible dialogue text enters that archive.

A dialogue record is not a filtered Event wearing its old name. Its source sequence points to where the text came from; it is not a newly allocated event sequence. The reducer never consumes `dialogue_log`. Preserving prior available dialogue records on subsequent saves keeps the archive useful, but switching back to full mode cannot restore the originals from that projection.

Dialogue-only mode still requires a non-null snapshot. It does not compact that snapshot, which may retain tool or provider material needed for current state. Nor may the policy filter the canonical recovery tail above `as_of`. That tail stays complete and separate in `log` until a newer snapshot folds it. A dialogue-only archive is consequently neither complete replay history nor a promise that all non-dialogue material has disappeared from the save.

Diagnostics have another purpose again. API and GUI traces explain operation; they are not the conversation's recovery state. Bill reports that his Gemini CLI fork ran out of disk space, though the source does not establish that fork's storage internals or whether it retained context snapshots. The report supplies a reason to care about disk growth, not a diagnosis of another system's persistence design.

The diagnostic policy retains the newest configured number of complete records in order. The `diagnostic_max_records` setting defaults to 200 per managed diagnostic log at each retention boundary. Zero means unlimited; negative limits are invalid. Counting lines would split multiline records, and counting bytes would be a different policy. These boundaries limit retained records after cleanup, not disk usage continuously between cleanups.

History policy belongs to the Agent; diagnostic cleanup belongs to the writer. Both settings use the existing settings ownership and persistence from Chapter 8, not provider credentials, browser display preferences, or extra fields in the recorded request configuration. An explicit zero must survive settings updates and restart. Unknown history modes are errors. The mode recorded in a save describes that file; current runtime settings govern the next save.

Cleanup comes after recovery is secure. At startup, load or recover successfully before trimming, establishing a valid recovery save first if none exists. At normal exit, finish accepted work and save successfully before trimming. A failed load or save leaves diagnostics intact, precisely when those diagnostics may be useful. Constructing a dialogue-only replacement likewise must not destroy prior recovery state or trim live history if the save fails.

The writer must coordinate retention with appends. Replacing a diagnostic file requires reopening it or resetting the writer's offset before writes resume. External truncation can leave a non-append descriptor positioned beyond the new end, creating a sparse gap. Renaming or unlinking a pathname does not close an open inode or guarantee that its space is reclaimed. Cleanup failure should be reported without destroying the prior diagnostic file or invalidating the recovery save.

This policy does not authorize a general disk sweep. Saves, recovery journals, memories, and worker state are not diagnostics. Shared rebuildable caches need separate authorization. Chapter 14 will add retention after a successful `micro_handoff` save; no compactor or handoff tool is introduced here.

Default Recovery Must Refuse Destructive Help

For the single-Agent commands, the ordinary location is `save.json` in the working directory. `--save PATH` changes the location for both loading and saving, not whether loading occurs. The library's default belongs under the owning Agent's data directory so two Agents do not accidentally share a process-wide conversation.

Path resolution carries the same ownership distinction. A relative `DataDir` follows its Agent's `BaseDir`, rather than whichever directory happened to launch the process. An explicit `SavePath` remains the caller's choice and must not be silently rebased. Preferences keep their separate persistence; conversation saving does not take it over.

Startup first validates the selected primary completely. If it is missing or invalid, startup independently validates its `.bak` sibling. Only two absent candidates mean a fresh conversation. If neither existing candidate is valid, startup refuses, names the failures, installs no restored state, makes no provider request, and leaves both files untouched.

The refusal is deliberately less friendly than a silent reset. A fresh session would eventually overwrite the damaged conversation with a perfectly valid account of having forgotten it.

A backup recovery must report the boundary it actually restored. Its snapshot, anchor, and log travel together. A loader cannot attach an arbitrary tail from the damaged primary and call the result recovered history. Here the log lives inside the save, so newer bytes lost with that file are lost to this mechanism. Chapter 14's separate journal will change what evidence is available after the backup anchor; it is not already present because the loader understands tails.

The timing of save registration matters too. Save-on-exit belongs after successful loading. An unconditional deferred save installed before `Load` can return exactly the right error and then overwrite the bad file on the way out. Checking the error alone would miss the destructive half of the behavior.

Successful saving first encodes and validates the entire new candidate without changing either file. If the current primary validates, its bytes become the backup through a temporary file and rename before the new primary is installed the same way. If recovery used a good backup because the primary was absent or corrupt, that backup stays intact while the new primary is installed. Corruption must never be promoted over the recovery copy.

The existing save owner serializes this operation for one Agent. Encoding, closing, or replacement failures propagate to the caller and leave a valid recovery candidate intact. They do not trigger diagnostic cleanup. Atomic replacement prevents a partially written JSON file from becoming the primary; it does not promise universal power-loss durability. Normal-exit persistence does not need a database or a write-ahead journal smuggled in under the word "atomic."

EOF Is the Beginning of Shutdown

Closing stdin ends admission of input, not necessarily the work already admitted. A final turn can still be running, and a quiet observer stream does not prove that it has finished. The application waits for reliable completion, shuts down the Agent, joins managed jobs that can append final facts, and only then saves.

Both CLI and GUI use that lifecycle behind the same public Agent. In the GUI command, closing stdin ends the application after admitted work is flushed; it is no longer merely detaching a terminal view while the server carries on. Browser prompts and terminal input still reach the same Agent.

The source's CLI emits final usage before its deferred shutdown save. That usage record therefore cannot acknowledge persistence. Successful process exit is the observable boundary before reopening the file. The same distinction runs through the chapter: finishing an answer, saving a snapshot, installing a replacement, and recovering it later are different events.

The source reports a live tool-bearing turn carried through EOF and relaunch, followed by an answer from the restored conversation. An initial paid attempt failed at the transport boundary; bounded TLS recovery completed the two launches without establishing the original failure's cause. That observation supports the tested model's restart path, not every provider or every failure mode.

Chapter 11: MCP: The Extension Protocol

Chapter 10 lets a conversation survive its process. It does not bring the old process's capabilities back to life. A saved tool result is still a fact about past work, not a connection to the program that did it.

MCP, the Model Context Protocol, introduces that connection as a separate responsibility. An Agent can discover tools in an external process or browser, admit them under a skill, and call them through its existing jobs. The peer can also call back into the Agent. What looks like a convenient way to avoid recompiling tools is therefore a new trust boundary and a new lifecycle to own.

The source chapter develops a client for the historical wire version `2024-11-05`, using JSON-RPC 2.0. Its reverse `tools/call`, `ephemeral` field, and `$/cancelRequest` notification are course extensions, not claims about what every current MCP server supports.

Discovery Is an Offer

The ordinary connection begins with `initialize`. The peer replies with its capabilities, the client sends `notifications/initialized`, and `tools/list` supplies

the available tools. Each offer has a name, description, and input schema. The bridge turns admitted offers into the same kind of tool definition the Agent already knows how to dispatch.

There is no need for a second execution system. When the model calls an admitted external tool, the existing job machinery invokes a handler that retains the peer connection and that tool's information. The handler sends `tools/call` with a string `name` and an object of `arguments`; the returned text blocks retain their order. A tool result marked `isError` is a failed tool outcome. A JSON-RPC error is a protocol failure. Converting either into a successful empty string would make the bridge easy to use by making its failures impossible to understand.

Before any of this becomes model-visible, the owning skill supplies the grant. Discovery says what the server offers; the skill says what this Agent may use from that server. Another loaded skill that happens to name the same tool must not widen this server's grant. The Agent's enabled-tool policy then applies at both declaration and dispatch, and an external definition cannot shadow a compiled capability with the same name. Checking only the declarations would leave a guessed name able to cross a boundary the model was merely encouraged not to notice.

The public embedding path follows the same rule. `Agent.MCP().Connect(ctx, name, transport)` completes initialization and discovery before exposing tools. With skills enabled, the peer name must identify a declared server. Supplying a transport is not a way to bypass admission.

That admission has a second consequence. The channel is bidirectional: a server can send its own `tools/call` request and reach enabled ordinary Agent tools such as `read_file` or `run_command`. The client dispatches through the Agent and returns a reply bearing the server's request ID. Unknown, disabled, and hidden automatic tools remain unavailable even if the peer guesses their names.

A server's offered-tool filter limits what it exports. It does not confine what the server can do through reverse calls. Connecting a peer means trusting it with the Agent capabilities enabled at that boundary; a grant for a remote search tool is not a filesystem sandbox for the process offering it. Confinement is later work.

One Channel, Several Owners

The transport carries complete JSON-RPC messages. An in-process pipe pair exercises the real client and codec without a subprocess. Stdio connects to a child process; a raw adapter wraps supplied streams; a WebSocket transport carries the same messages through the GUI to a browser.

Those variants share a small boundary: send one complete message, receive one complete message, and close. One receiver consumes the stream while concurrent senders may produce requests and reverse replies. Whole-message serialization matters because those sends can overlap.

The common transport interface and its public alias let external clients connect without importing the GUI. Raw streams, pipes, and subprocess behavior belong to the MCP package. WebSocket framing belongs to the separate GUI module. A headless application should not acquire browser machinery to call a tool in a Python process.

Ownership follows the runtime rather than a new collection of sibling service pointers. The Agent owns the MCP manager, which owns its clients. A client reaches admission, jobs, and diagnostics through manager, Agent, and Ensemble. A GUI transport retains its actual server parent. The composition root establishes those relationships; a reader and writer are boundary resources, not reasons to invent another domain object.

Within a client, one owner can hold the transport, incrementing integer request IDs, pending-reply map, and reader. Replies are matched by ID and delivered to the appropriate waiting caller. The source's accepted design keeps correlation there rather than exporting a separate codec with no other user.

Reverse requests make the reader's job particularly important. It must arrange execution without becoming the executor. A reverse tool can itself call the peer; if the handler occupies the sole reader, its nested call waits for a response that nobody is free to read. The peer can have answered perfectly and both sides can still wait forever. Keeping the reader available also permits out-of-order replies, which must reach their original callers rather than whichever goroutine happens to be waiting first.

Closing this boundary must wake blocked sends and receives. For a subprocess, closure includes its pipes, process group, and reaping the child. Closing only a Go wrapper can leave an external server alive after the skill that authorized it has disappeared. The lifecycle is not complete until the resources on the other side have been dealt with.

The Job Owns the Call

The bridge has an easy-to-miss line of responsibility: the outstanding RPC uses the job's context. The original example used `context.Background()` instead. That version can return every short successful call correctly while making explicit kill unable to reach the call still waiting on the wire.

In the chapter's design, killing the job cancels its context and sends the peer `$/cancelRequest` with the outstanding request ID. The notification belongs to this wire contract; sending it is

not, by itself, evidence that an arbitrary server has stopped its work. The Agent still has to release its own waiter and manage its owned resources.

A callback deadline means something else. It returns control with a job handle while the work continues. Interrupt ends the current turn without killing that job. Shutdown stops owned work and closes peers. These are four different decisions, even when they occur during the same slow RPC. Giving them one context would make an observation timeout cancel useful work, or make shutdown wait on work it can no longer control.

Reverse calls use those real jobs too. They leave execution facts in history without fabricating a model-authored tool call merely because the peer used the same method name. A reverse caller may legitimately receive a running-job notice and follow up. The next mechanism has no such caller, and that difference exposed a particularly revealing failure.

A Hidden Tool Has Nobody to Ask Again

Some observations should be collected before the model decides anything. A peer can mark a discovered tool `ephemeral: round` for a fresh observation before every `RequestSent`, or `ephemeral: turn` for information collected once at the start of each new turn. The owning skill must grant these tools too. Automatic execution is not an exemption from admission.

The model neither sees their declarations nor calls them. Their completed results enter `Context.Ephemera` through the existing event and reducer path, to be consumed by the next request. The event log retains the arrival for replay, while stale observations do not pile up as dialogue. The turn-scoped call runs once for that turn; it is not a license to reuse its data in the next one.

Hiding the ordinary declaration is only the beginning. Chapter 9's deferred catalogue and skill-loading `tool_reference` blocks can also be built from the capability snapshot. Automatic names must stay out of all those provider-facing surfaces and remain unavailable to guessed ordinary calls. The internal snapshot, however, must retain them so the engine can collect them. Deleting them from the registry would solve disclosure by disabling the feature.

The source records a first implementation that gathered automatic context using an ordinary job wait. An independent peer delayed its turn snapshot by 3.4 seconds. At three seconds the callback returned a running notice containing `Use wait_for_job`, and the provider request carried that notice where the snapshot belonged. The once-per-turn flag was already set. The actual result arrived after the request that needed it.

Every local component had done something plausible. The job kept running. The callback returned promptly. The engine attached the returned text. What failed was the assumption that returned text necessarily meant completed observation. A hidden tool has no model-directed caller who can see a handle and ask again. Increasing the timeout would merely move the failure to a slower peer.

The correction preserves the ordinary callback path while giving the automatic collector the handler's completed result through a buffered reply. The provider request waits for actual content or an actual failure. An observation failure is logged but does not abort the turn; a running notice is neither the observation nor its failure.

Waiting must not freeze the Agent. Collection happens outside the actor so reverse requests, interruption, and job facts can still be processed. The completed collection returns with its turn identity and attaches only if that turn is still active. Interrupt can

therefore abandon the collection for the old turn without killing the underlying job. The job's eventual output remains available, but it does not become fresh context for a different turn.

The source's regression deliberately held an old peer reply, interrupted, completed a new turn, and then released the old result. Another check held both round and turn replies beyond the callback deadline and looked for an early provider request. These tests used the real jobs, actor, reducer, and renderer with substituted external peers. They probe the relevant ordering, not every possible server or provider. The important result is a distinction the fast path conceals: completion gates a request, while turn identity decides whether that completion still belongs to it.

The Browser Is an On-Demand Peer

A browser cannot provide the same subprocess boundary. The GUI already carries prompt, hint, interrupt, and settings traffic over `/ws`; an additional `jsonrpc` envelope tunnels MCP messages through that connection. Replies retain their JSON-RPC IDs. Each peer needs reliable delivery through its connection's serialized writer. A lossy progress broadcast is acceptable for some views of activity, not for the only copy of an awaited RPC reply.

The implementation described in the source selects one active browser MCP target. Other tabs retain ordinary chat and settings access rather than becoming competing recipients of a tool call. Browser discovery needs both a declared skill and a connected browser; an already-open browser can become the peer when its skill is loaded later.

The browser module offers four ordinary tools. `gui_snapshot` walks the visible DOM and returns a Markdown account of pane layout, interactive elements and their CSS selectors, and artifact previews. `gui_click` dispatches a click on a matched element. `gui_input` sets a value and dispatches input and change events. `tts_queue` returns pending speech text, state, and timing as JSON.

The snapshot is bounded at 4096 UTF-8 bytes, but the missing material must not disappear silently. It reports elided characters and omitted artifacts, and includes the state attributes of every control it lists. A selector without state may identify exactly the right control while leaving the Agent to guess whether it is checked, disabled, or already set. A truncated snapshot without disclosure can turn an incomplete view into an unwarranted claim that the screen looks fine.

The intended interaction is observation, action, then observation again: inspect a visible control, use its selector, and obtain a fresh snapshot to check what changed. Speech-queue inspection is similarly explicit when pending speech or timing matters. Queue state is not evidence that a human heard intelligible audio. That requires listening.

Earlier browser observations were automatic. The source reports that the crossover found per-round snapshots and speech-queue checks noisy and unnecessary, so the revised design makes all four tools ordinary admitted declarations. This is a reason to change the scheduling of those observations, not to remove automatic context collection generally. A changing environment may justify a fresh observation before every decision; a browser snapshot is useful when the work needs a browser snapshot. Both consume work and attention, so the distinction belongs in the capability's design.

A Skill Starts More Than Instructions

Skill frontmatter names each server and its transport. Stdio supplies a command and argument array; WebSocket selects the browser tunnel. Loading the skill starts its servers, discovers their offers, and applies its grants. For a code-search server, for example, the skill can authorize `search_code` without admitting every capability the process advertises.

Arguments are literal strings, not shell fragments. Relative script paths resolve from the Agent's workspace. Provider credentials remain with the model client; the subprocess launcher described in the source passes only basic operating-system environment values to external servers. Avoiding shell expansion and credential inheritance narrows accidental exposure, but neither establishes a sandbox for the peer. The frontmatter parser handles the declared fields without taking on a general YAML implementation.

Unloading removes that skill's external executable tools and stops its servers immediately. This deliberately differs from Chapter 9's pending removal of compiled tools and retained instruction text. A manual can remain in context without consuming a process or retaining an external connection. Keeping the manual is not a reason to keep the server alive, and stopping the server does not imply that the manual has been erased. Compaction still belongs to later work.

This also completes the distinction from persistence. Conversation state can remember that a tool was used, and retained instructions can describe how it worked. Neither fact starts its server, reestablishes trust, or makes its handler executable. Those are decisions of the current Agent and its current skills.

Follow the Result, Not the Appearance of Success

A dedicated `--mcp-pipe` service reserves stdin and stdout for newline-delimited JSON-RPC objects, keeps diagnostics off stdout, and initiates the handshake without starting a conversation. It can establish that two peers agree on framing and initialization. It cannot establish model-driven tool execution because there is no model turn in that mode.

The source's small demonstration peer makes the next boundary visible. An ordinary tool reads a marker file, a round tool supplies a changing snapshot, and a turn tool supplies a stable session brief. Local call counts allow comparison with actual provider requests rather than reliance on the model's account. The round observation can run before the initial request and again before the request carrying the ordinary tool's result; extra model rounds can add calls. An expected count is not evidence of the rounds that actually occurred. The demonstration peer serves only its listed tools and does not test cancellation.

Historical Claude, OpenAI, and Gemini receipts record successful runs of this pattern. They are evidence of those runs, not current model availability or comprehensive provider coverage. The first Claude attempt is more instructive than a list of successes: it exited zero without producing an answer. Its parser had replaced valid initial `{}` arguments with an empty `input_json_delta`. Keeping the initial input when no JSON bytes arrived fixed the call. A successful repeat captured the empty delta, and a regression reproduced the failure when the defect was restored. Protocol completion, process success, and a useful answer had been three different facts.

A separate bridge test initially accepted deliberately broken argument forwarding. Its assertion searched the whole provider request for the marker, so it found the marker in the earlier call's arguments even though the returned result was wrong. Restricting the assertion to the correlated result blocks exposed the defect. A request can contain both what the model asked for and what the tool actually did. Evidence for one cannot be borrowed from the other.

The browser evidence has its own boundary. The source's historical receipt used real Chrome, the Go tunnel, and the Agent with a local provider fake. It observed control changes, native speech start, and disclosed snapshot truncation, but exercised the old automatic observations rather than the revised on-demand path. Applied font state did not prove persistence, and native speech start did not prove intelligible audio. It was not a live-provider browser demonstration.

The source also records earlier checks that credited tool calls and ephemeral behavior without executing them, and a WebSocket check that searched fixed filenames instead of running a tunnel. Later reports do not retroactively make those checks adequate or establish Edition 3 acceptance.

Chapter 12: The Agent Sees Itself

Chapter 11 gives an Agent a connection to browser tools. That does not yet give it a reliable account of its own interface. A click can succeed while the screen stays wrong. A snapshot can describe every button and omit which one is selected. An observer can even stop the system it came to inspect.

Bill's experiment with a GUI-only driver made those distinctions unusually concrete. The driver could not read source, edit files, or run commands. It had to use the interface, much as a human would. Its first attempt to type a prompt froze the coding agent before the task began. Later it reported a Settings panel that did not exist. Between those failures, it helped expose defects that source development and passing checks had missed.

The source chapter records that history alongside a smaller startup design: select one skill and reuse the browser connection already built. The plumbing deserves to stay small. The judgment required to use its results does not.

Selecting the View at Startup

Both the headless command and the optional GUI accept `--gui-debug` and `--skills-dir PATH`. The first selects the reachable `gui-debug` skill during Agent construction; the second overrides `EN_SKILLS_DIR`. Debugging without a selected directory uses `./skills`. When a directory is selected

and `EN_PRIMARY_SKILL` is empty, these applications choose `ensemble` as the primary. These are application defaults, not reasons for the core to acquire GUI dependencies.

The primary and optional skill make different promises. The `ensemble` primary grants ordinary coding tools and lists `gui-debug` as loadable. The latter is explicitly a loadable skill, depends on `ensemble`, and grants `gui_snapshot`, `tts_queue`, `gui_click`, and `gui_input`. Its `browser-debug` MCP server uses the WebSocket tunnel. Kind determines how a skill participates, reachability determines whether it can be selected, dependency supplies the base, and the tool names determine admission. None can substitute for another. Discovering an impressive extra tool does not grant it, and reverse calls still pass through the Agent's enabled-tool dispatch.

The application expresses the startup choice through `Config.StartupSkills`; construction loads the primary and activates `gui-debug` through the same admission path as a later load. Both manuals enter initial SYSTEM. There is no model-authored `load_skill` call to answer, so a fabricated tool result would be fabricated history. The implementation described in the source records application-origin activation at sequence zero instead.

This stays inside the existing ownership tree. The Agent owns its registry and MCP manager; the registry reaches the manager through its actual parent interface. The composition root attaches those children before loading skills. The GUI supplies its public transport and owns browser framing, while its executable consumes the public core. A new callback bundle, second dispatcher, or permanent driver application would mostly create new places to get these relationships wrong.

Startup has one genuine timing wrinkle: the bundled browser connects after the Agent is constructed. Known metadata for its four tools permits admission before attachment, not execution

without a browser. An already-connected peer supplies its actual catalog instead; cold-start metadata must not replace discovery. Configured endpoint and transport overrides remain meaningful. When the real peer attaches, its schemas and ordinary-versus-automatic identities are validated, and the owning skill's explicit names still govern permission. An unknown required name is an admission failure, not an invitation to guess a capability into existence.

Later loading adds instructions through Chapter 9's provider-aware SYSTEM path, with the paired tool result reporting activation rather than becoming instruction authority. No cache-hit promise follows from that arrangement. Unloading has the split lifetime established in Chapter 11: external grants disappear immediately, peers with no remaining grants close, and a stdio peer is terminated and reaped while the Agent can continue. Guessed calls must fail too. Withdrawn external declarations disappear even from Anthropic's deferred catalog; ordinary dormant compiled tools retain Chapter 9's disclosure behavior. Compiled grants and retained manuals remain subject to pending removal. Compaction is still future work, not a side effect of closing a browser connection.

The operational distinction matters even before a test starts. A headless process can use a stdio peer, but the actual browser needs the separate GUI application and a connected tab. Its web assets come from disk, so the asset directory must match the launch location. The debug flag supplies capabilities, not an implicit prompt. Headless protocol input remains newline-delimited, EOF drains submitted work, and the interactive `chat` command follows its flags. Chapter 10's default conversation loading also still applies; an apparently fresh browser session can be an old conversation unless its saved state was deliberately separated.

Observation Is a Call, Not a Background Assumption

All four browser tools are ordinary on-demand tools. `gui_snapshot` returns the DOM account: panes, selectors, semantic control state, and artifact previews. `tts_queue` returns pending utterances with their text, state, and timing. `gui_click` acts on a visible selector; `gui_input` sets the named control and dispatches input and change events. Chapter 11's automatic-context mechanism remains useful elsewhere, but browser snapshots and queue inspections are not attached to every request.

The ordinary sequence is an observation, an action using what was observed, and a new observation. In the supplied browser, `ok` from a click or input establishes that the handler ran. It does not establish that Settings opened, that a value survived validation, or that the effective page font changed. The next snapshot has to reflect the interaction. A running-job notice is not a delayed DOM rendered in an unusual format; the observer needs the actual result.

This also determines where evidence is read. Searching an entire provider request for the desired value can find the value in the earlier call's arguments. That proves what was requested. The correlated result and refreshed state show what happened. Requested speech-queue contents likewise establish pending text and status, not that a listener heard intelligible audio.

A stateful stdio peer is useful here because it substitutes only the external transport boundary. It can retain the same skill kinds and grants, exercise discovery and dispatch, change state after an interaction, and reveal unsolicited observations or failed cleanup. It cannot prove semantic state in a real DOM, native speech behavior, or delivery through the browser tunnel. The historical

wire version and course extensions retain Chapter 11's scope; changing transport does not make them a universal MCP guarantee.

A Driver That Cannot Peek

The source attributes the first-edition experiment to a retained experience report. Its driver was a separate small binary built on `NewBareAgent`, with no builtin filesystem, shell, or editing tools. It connected to the browser's hub, discovered interaction tools through MCP, and supplied `file_report` as a place to put its conclusions. That historical driver also gained submission and waiting helpers. Those helpers are not additions to this chapter's four-tool startup design.

Removing source access changed the question the experiment could answer. A coding agent can explain a button by finding its handler, which is useful when repairing the button and inadequate when establishing what the user experienced. The restricted observer had to obtain its answer through the interface. A prompt telling a fully equipped agent not to read files is a weaker boundary than having no file tool to call.

Bill supervised while reading the streamed reasoning. He let the driver inspect before supplying a diagnosis, because an observer that walks past a defect is revealing something about its view. Pointing at the defect first would erase that evidence. He still supplied operational direction: what to exercise next, which binary to rebuild, and, on one occasion, that a stale server held port 8084 while the new binary was serving nobody. There is little experimental value in letting a driver spend its effort on the wrong process.

Findings need a separate attribution. The report credits Bill with a screenshot observation that led to the truncation investigation, and credits later source inspection with finding unsafe rendering. Neither becomes an unaided GUI discovery because an agent performed the repair. Keeping assistance attached to the result lets a successor judge both the tool's reach and the human work still needed.

Typing Without Becoming a Human

The first driver runs stopped at the prompt field. Typing caused the coding agent to pause permanently.

The pause feature existed for a good reason. Bill reads reasoning as it streams and sends corrections mid-turn. An engine that keeps deciding while he is composing a hint can make the decision before the correction arrives. The original browser listener therefore paused when nonblank input appeared and unpaused when the field became empty.

The driver followed a different event sequence. `gui_input` set the value and dispatched an input event, so the pause fired. The historical `gui_submit` then sent the prompt and cleared the field by assigning to its `value`. JavaScript assignment does not dispatch another input event. The unpause branch depended on an event that never came. The GUI had observed a user start typing and, as far as its listener knew, continue forever.

The first repair marked programmatic input so the listener could ignore it and made submission explicitly unpause instead of relying on an implied event. That fixed the immediate path. It did not yet preserve the reason for pausing throughout the browser.

In the second-edition work, a real-browser check showed that the model could fill the field and click Send with speech disabled. Independent review enabled native speech. When the utterance finished, programmatically supplied text was still in the field and the Agent stayed paused. The speech callback called `pace()`, which inspected the field's contents anew. The input listener knew the text was programmatic; the later callback silently reclassified it as human typing.

The durable correction keeps the cause rather than reconstructing it from the value. MCP input dispatches a bubbling input event marked with `detail.mcp`, followed by change. The listener updates an explicit `humanTyping` state: unmarked input with nonempty contents represents human typing; marked input does not. Speech and typing combine as `speaking || humanTyping`. Successful submission clears `humanTyping` explicitly after clearing the field. Human whitespace counts as typing. Programmatic text does not become human merely by remaining visible until speech ends.

That distinction has a second boundary: pause belongs to the whole Agent, not to a tab. A browser sends automatic pause and unpause commands as its local causes change. If another tab explicitly unpauses, an unchanged local cause must not immediately reassert pause and veto the human's decision. Local pacing cannot quietly promote itself to permanent authority over the Agent.

The corrected native-browser check covered speech ending while programmatic text remained, genuine whitespace holding pause, explicit unpause from another tab, and a later actual MCP submission. It waited for the concluding speech to finish. Those checks used the real browser and Agent with a local provider fake. Native start/end events establish the measured transitions, not listening quality. The second tab closed after the global-unpause

control, and an earlier two-tab speech stall has no established cause. This is evidence for the corrected paths, not a blanket guarantee about concurrent speech.

The failure generalizes beyond typing. Human-facing pacing, debounce, idle detection, and animations all encode assumptions about the cause and timing of events. An automated user can violate those assumptions while producing perfectly ordinary-looking field values. Ignoring one synthetic event is insufficient when another callback later invents its origin again.

The Escape Hatch Behind the Wait

Once input worked, the historical driver asked the coding agent to write Tower of Hanoi and debug it. The agent produced fifteen moves, started `dlv`, and hung. The book had already warned about precisely this trap.

The command tool synchronously read the PTY until output closed. A short `go build` made that lifetime look harmless. A debugger waiting at its prompt kept the PTY open, so the tool function never returned. The callback pattern could have recognized the prompt, except that pattern matching happened inside `Wait`, and `Wait` was reached only after the tool function returned. The escape hatch was on the far side of the blockage.

The repair separated starting the work from owning its completion. The tool returned once the process was running; a reader goroutine retained the PTY, copied output into the job, reaped the process, and finished the job at exit. The historical dispatcher used a `DeferFinish` field for that handoff. It is an explanation of the old repair, not a new API to introduce now. Chapter 4 already establishes the independent job lifetime that browser access must preserve.

Then the debugger worked and the older Chapter 4 checks failed. Returned tool output was empty, and the capture file contained zero bytes. The apparent output defect was another ownership defect: that codebase had two dispatchers. `Actor.dispatchTool` knew about deferred completion; `Engine.Execute`, used by the older `Ask` path, still finished the job immediately when the tool returned. Finishing closed the capture file while the reader goroutine was writing to it.

Debug prints in the actor dispatcher never appeared. The first suspicion was a stale build cache. The failing path did not call that dispatcher at all. Before rebuilding the world, it is worth finding out whether execution visits the line being instrumented.

With both dispatchers corrected, the report records a complete rerun: a three-disk program produced seven moves, then the agent set a breakpoint on `move`, continued, printed `disk`, and quit the debugger. The report gives a zero debugger exit and ninety-eight seconds elapsed. The earlier hung run used four disks and fifteen moves; they are not one increasingly successful measurement. This is a reported historical outcome, not a new benchmark or evidence of the present checkout.

Correct on the Server, Wrong on the Screen

The settings test first found the obvious defect. The driver entered a temperature of negative five and the server accepted it. Clamping incoming patches and settings loaded from disk closed both entry paths, including a hand-edited settings file.

Repeating the same interaction exposed the more interesting defect. The server now clamped correctly, but the screen still displayed negative five.

One `Settings` struct was doing two jobs with incompatible meanings for omission. In a sparse patch, an absent field means leave the existing value alone. In a complete broadcast, every field must be represented, including zero and false. The struct used `omitempty`, so the temperature clamped to zero disappeared from the broadcast. The browser updated only fields present in the message and therefore retained the rejected value. The same encoding omitted `TTSEnabled: false`: turning speech off could not be represented in that full-state message.

A comment in `SettingsStore.Apply` acknowledged the false-value problem and routed around it. The method had no callers outside tests. It was evidence that somebody had encountered the defect, not that the state model had been repaired.

There were two reasonable directions depending on compatibility. Separate patch and broadcast types could preserve a wire format with external consumers. In this case the maintainer established that there were none, so preserving that format would retain complexity without protecting anyone. The historical repair removed `omitempty` from the broadcast fields and deleted the unused `Apply`, leaving one merge path with defined semantics. The lesson is not that two types are inherently bad; it is that compatibility should protect an actual dependency rather than an imagined one.

A complete state broadcast must carry `temperature: 0` and `tts_enabled: false` explicitly. Even then, the stored value, the server's applied value, and the client's displayed value are separate observations. A correct file on disk does not prove the screen. Chapter 8's corrected account follows all three. The source also reports a historical check with clamping disabled, but that report is distinct from this edition's settings evidence.

The Panel That Wasn't There

Asked to exercise sidebar tabs, the driver confidently reported that clicking Artifacts opened Settings. There was no Settings tab in that version of the markup. The tab-to-panel mapping was correct.

The interface was hiding the fact the driver needed. Selected state lived in CSS classes, while `gui_snapshot` built labels from text, `aria-label`, and placeholders without reporting state attributes. The observer could see that tabs existed but could not see which was selected. It also used the ambiguous selector `button:nth-of-type(2)`, because the controls offered little better to aim at. Faced with a request to say what changed, it supplied a plausible panel instead of an observation.

The repair made state explicit. The hamburger exposed `aria-expanded`; tabs exposed `aria-selected`; click handlers kept those values synchronized. Initial state came from the live CSS class rather than a hardcoded default, because markup and rendered defaults had already drifted. The snapshot included the state beside each control.

The historical report says that repeating the same test with the same model and prompt then produced correct before-and-after state and used "not observable" where information was still absent. That is a useful result, not a law that missing information forces every model to invent an answer. In this run, hiding state helped produce a false report. Exposing state supplied the evidence for a correct one.

Accessibility semantics are consequently part of the debugging interface, not decorative metadata. A selector gives an observer somewhere to act. A state attribute gives it something to learn afterward. Both have to describe the live interface rather than the intended default.

Two nearby findings sharpened the distinction between a useful view and a trustworthy one. Source inspection found the tool-card header interpolating tool names and serialized arguments into `innerHTML`, even though the result path nearby used safe `textContent`. Arguments can contain filenames, URLs, and file contents influenced by an attacker. Their arrival through a tool call does not make them trusted markup. The unsafe path needed a rendering repair; the browser driver's other findings did not prove it could discover that path independently.

Bill's screenshot observation prompted the truncation investigation. The initial notes claimed the driver saw the full DOM while the human saw a trimmed view. Measurement found the opposite imbalance. The historical human view capped tool input at 500 characters and results at 1000; the snapshot capped each artifact at 120 and the whole document at 4000, then silently omitted every artifact after the tenth. It could not distinguish ten artifacts from fifty.

Those are historical caps, not a replacement for Chapter 11's revised 4096-UTF-8-byte snapshot boundary. The enduring repair kept bounded views but made their omissions legible. The historical interface retained truncated text through a `title` attribute, and the snapshot reported withheld characters and artifacts. Limits protect attention and context; silent limits manufacture confidence about material the observer has never received.

What the Later Runs Actually Established

The source records later real-browser sessions with the historically exercised `claude-haiku-5-5`, `gpt-4o-mini`, and `gemini-3.8-flash`. These names identify the recorded sessions, not today's model catalog.

Claude Haiku and Gemini changed the effective font to 20, received a refreshed snapshot, and unloaded the skill. Claude then said the MCP tools remained available until compaction. The final captured request already showed them gone. An old generic unload message had encouraged the wrong explanation, so the message was clarified to distinguish immediate external removal from pending instruction removal. A wording repair did not require another paid run to establish the already-observed removal.

Both GPT-4o mini attempts failed the full task. The first guessed `input[type='number']`, received a missing-selector error, and unloaded without recovering. In a second, guided attempt, it was given the observed `#font-size` selector and told the control was a select. It changed the setting and unloaded, then attempted to read `path/to/the/snapshot` despite an instruction forbidding file access. The nonexistent path failed, and the final response claimed the snapshot could not be inspected, although the captured interaction already contained its refreshed state.

That second attempt passed the script's GUI assertions. It still failed the task. The controls worked; the model mishandled the evidence and violated the requested boundary. The supplied selector was assistance, and a prompt-level prohibition did not provide the confinement of the earlier bare driver.

Those Edition 2 sessions preceded the final native-speech correction and small manual and message corrections. They establish their recorded interactions on their captured source, not a paid rerun of the final source. Speech has its separate native-browser evidence. Effective font size is also not restart persistence.

The source separately records an accepted Edition 3 run that changed font size from 16 to 20 using selectors observed in snapshots. A fresh snapshot and computed styles confirmed the effect, not pixels. After unload, the next provider request omitted all four browser declarations while retaining the manual. The GUI displayed the final response and saved on normal EOF exit. That run did not repeat native speech testing. None of these records is new implementation verification performed for this companion.

Earlier checks have narrower meanings too. The original Chapter 12 grader returned zero after an untyped fixture blocked startup, while its parity detector depended on particular method names. It observed neither successful browser behavior nor the later speech/input failure. The corrected obligations instead follow actual state through both clients, correlated snapshot and queue results, absence of unsolicited observations, interaction, unload refusal, and peer cleanup. Existing command lifetimes remain relevant: the inherited two 62-second header/body waits must finish against the same executable before a compatibility claim can be made. A short timeout is missing evidence, not an abbreviated pass.

The Book Gets a Bug Report

The historical repairs traveled back into the chapters that should have made their obligations clear. Explaining an interactive job model had not been enough to prevent a tool from waiting for process exit. Showing a tool card had not identified the attacker-influenced text it rendered. Specifying a snapshot cap had not said that the cap must announce its omissions.

The old settings material was especially revealing. It treated the graded surface as a handful of server checks and left client behavior as something visible on screen. But seeing a value accepted does not establish validation, seeing the server's corrected value does not establish the browser's, and a human-readable tab does not establish machine-readable selected state. Several defects fit inside those unstated obligations.

That does not make every implementation mistake the book's fault. It does make the instructions worth repairing when an experienced implementer could have inferred the missing duty and did not. The experiment produced changes to both the software and the teaching that would shape the next implementation. It is a concrete feedback loop, not evidence of unrestricted autonomous self-improvement.

Even the account of that loop needed correction. The first-edition chapter said the sentence “You know the client works because you can see it.” had been invented and was absent from the original settings chapter. The source chapter reports that the exact words were there, split across a newline, including in the revision before the narrative was appended. A one-line search had missed a wrapped sentence.

Chapter 13: The Channel Nobody Tested

Bill listens to the coding agent for hours a day. In the first-edition investigation, that dependence produced a suspicion rather than a tidy bug report:

> *I'm worried the TTS feedback, available via the MCP tunnel, isn't evaluated by any agent. It needs work.*

Chapter 12's GUI observer could read a snapshot, complete a task, and never notice that the same information had failed to reach speech. Giving it another observation tool would help only if the tool exposed useful evidence and the task actually depended on that evidence. Otherwise the observer could keep succeeding by looking.

The source chapter takes the more revealing route: remove the DOM. A listener that can obtain the target's answer only through speech has a reason to notice silence. That does not make it a complete accessibility evaluator, or even a judge of how a voice sounds. It makes a neglected channel part of the task's critical path.

Telemetry With Nothing to Say

The historical investigation found four paths meeting in the artifact renderer. Text and thinking deltas entered as they arrived. Tool dispatch announced itself. A completed part flushed the buffer. Errors rendered on screen.

The error path never called speech. It created a div, set its text, appended it, scrolled, and returned. The listener could miss precisely the message that explained why progress had stopped.

There was already a tool for inspecting speech: `tts_queue`. According to the source's account of the inspection, it read `window._ttsQueue`, but nothing wrote that variable. Its fallback returned an empty array or one synthetic entry containing `(speaking)`. The tool was marked `ephemeral: "round"`, so this contentless telemetry reached the model automatically on every round. The older GUI-skill check confirmed that injection worked. It had established reliable delivery of almost no information.

That is the historical mechanism, not the current four-tool GUI design from Chapter 12, where speech-queue inspection is on demand. The useful distinction survives either delivery policy: confirming that a result enters context does not establish that the result describes the channel.

Pause integration had a similar shape. The intended gate combined speech and human typing. Only typing was wired. The Go side checked pause before tool dispatch, but the browser never supplied the speech cause. Its speech module's opening comment advertised “Chrome wake-up and pause integration”; the reported inspection found no other occurrence of `pause` in the file. The comment had completed the integration before the code did.

These were not defects a daily listener could necessarily enumerate. Missing errors produce no bad utterance to examine. A missing pause may appear only as a tool starting too soon. An odd break in a sentence might come from segmentation, network timing, or the voice. Dependence on a channel makes its failures consequential; it does not give the user an independent record of what should have arrived.

The Correct Answer From the Wrong Path

The first blind-persona experiment was expected to fail. It asked the target to run a command against a nonexistent path and report the exit code. Tool results did not enter automatic speech, so the prediction was that the listener would hear the dispatch announcement and lack the answer.

It reported the correct exit code.

The retained transcript explained why. Its six entries began with “run command,” followed by “The command failed with exit code 1:”, a lone backtick, the missing-path error, another lone backtick, and the start of an explanation. The first entry announced dispatch. Everything afterward was the model's prose about the result. The raw tool result had never been spoken.

The listener accurately reported what it received and concluded that no gap had been found. That conclusion exceeded the evidence. Words containing an exit code cannot, by themselves, establish which route delivered it. A talkative model had bridged a gap in the experiment's prediction.

Bill's scope decision matters here. The intended interface does **not** automatically read successful tool-result bodies aloud: “I listen to your thinking, and that is enough.” Thinking and response text carry the running commentary. A person can

deliberately invoke an artifact's speaker button when its contents are wanted. Failed tools are different: the current dispatch path feeds their failures into error speech. Suppressing a successful file's contents must not also suppress the explanation that the file could not be read.

This changes what the listener is there to establish. Hearing the exit code in model prose is permitted behavior, not evidence that raw output was delivered. The question is whether thinking and response text arrive completely and in order, alongside identifiable announcements and errors. Source labels distinguish those paths for a program. Audible labels distinguish them for a person; the browser described in the source prefixes thinking and error segments with “Thinking:” and “Error:” before both recording and playback.

The policy has a tradeoff worth keeping visible. It avoids turning every file read into an unsolicited audiobook. It also means the listener's understanding of a successful tool result depends on what the model chooses to explain. That is the chosen commentary interface, not a promise to vocalize every byte available to the target. A listener that hunts for intentionally unspoken successful results is testing the wrong requirement.

The two isolated backticks, however, were nobody's intended interface.

Preserve Structure Until It Has Done Its Job

Streaming was the obvious suspect. Perhaps a fence had arrived across several deltas, leaving the filter with pieces that could not match. The historical investigation fed the entire fenced block as one chunk. Backticks still reached speech.

That counterexample moved the fault upstream. The old pipeline split the buffer on newlines, then filtered each resulting phrase. A fenced block contains newlines. By the time the fence pattern ran, its opening marker, body, and closing marker had become separate inputs. No amount of improving the pattern would reunite them.

The repair resolves fenced blocks against the whole buffer before splitting it. A completed block becomes a spoken name such as “code block,” rather than a recital of its syntax. An unfinished fence stays buffered while its structure is unresolved. Chunk boundaries are transport accidents; they cannot decide which syntax exists.

Newlines needed a second distinction. A single newline can wrap a sentence without ending it. Treating every wrap as a phrase boundary adds a pause the author did not intend. A lone newline therefore becomes whitespace, while a blank line remains a boundary. Sentence-ending periods also let complete phrases leave the buffer. End of stream, a tool announcement, or an error forces a flush, so a final tail ending in a colon is not left waiting forever for punctuation that will never arrive.

Filtering removes emphasis markers, inline-code markers, heading and bullet prefixes, and link syntax before speech. The order matters beyond fences. An early review found that flattening a newline before removing bullets allowed the second item's prefix to escape a line-anchored pattern. A perfectly ordinary two-item list exposed the mistake. Structure should disappear only after every operation that needs it has finished.

Cleanup can destroy paragraph boundaries too. The current `gui/speech_text.go` described in the source uses horizontal whitespace, `[ \t]`, in prefix-removal patterns rather than `\s`. The broader class can consume blank lines. Without that distinction, a whole final could merge paragraphs that happened

to stay separate during streaming. Identical text should not acquire different phrasing because the provider delivered it in a different number of pieces.

Identifiers get their own treatment: `camelCase`, `snake_case`, and `HTTPServer` expand into words, without chopping ordinary words or ordinary capitalization. That improves the synthesizer's input, but intelligibility still depends on the voice and listener. Expanding `HTTPServer` is a deterministic formatting decision; deciding whether its pronunciation is comfortable is not.

The buffer consequently has more responsibility than collecting characters until a timer expires. Deltas can arrive mid-word. The formatter must retain unfinished sentences, identifiers, links, and fences until a stable prefix can be emitted. Speaking each incoming fragment sacrifices meaning to network timing. Buffering forever preserves meaning for an audience that never hears it. Phrase boundaries and explicit flushes balance those two failures.

A Final Is Not Necessarily a New Utterance

Another historical path displayed a complete part without speaking it. No deltas had preceded the final. The handler set the card's contents and called `flush()`, which dutifully emptied an already empty buffer.

Queuing every final would repair that silence and make streamed responses speak twice. The current design instead gives each part its own `speechText` formatter, with an offset into the

raw source text. Filtering may remove many characters, so an offset into the cleaned text would lose its relationship to the next update.

For a delta, the formatter exposes only a stable prefix beyond the consumed offset. For a final, it exposes the remaining raw suffix. If no deltas arrived, that suffix is the whole part. After streaming, it is just the unsaid tail. A repeated final finds nothing new. Keeping state per part also prevents the unfinished tail of thinking from being joined to a response merely because the two arrived close together.

The source places this work in the GUI server. `gui/hub.go` feeds the formatters; `gui/speech.go` schedules segments with their source and part identity. The browser claims a speech task and admits its segments for playback. Replaying history reconstructs displayed cards without feeding old text into speech again. This is the current division of ownership, not the earlier browser formatter with its `card.streamed` guard.

That move changes where a useful final-only check must enter. In the historical implementation, turning provider streaming off reached the complete-part path. The current Engine can synthesize display deltas from a plain provider response. Disabling HTTP streaming therefore does not establish that the formatter received a final alone.

The previous edition caught this with a mutation: removing whole-final speech still passed the nonstream check. Suppressing deltas at the external WebSocket boundary then reached the browser-owned formatter. With formatting now in the GUI server, that boundary is too late. A meaningful check must deliver a complete part without preceding deltas to the actual formatter owner and observe the resulting speech.

This is also where a listener needs an expectation from outside the channel. It can notice that no reply arrived, but silence does not reveal which text was omitted, or whether the target

produced text at all. Comparing the emitted final with the transcript distinguishes missing speech from an absent answer. The restricted listener and the independent comparison answer different questions; neither needs to pretend to be the other.

The Transcript Begins Where Speech Begins

A transcript made from all queued text would look wonderfully complete even after cancellation discarded half the queue. It would describe what the application intended to say, not what entered the listener's channel.

The browser therefore records each filtered segment at the shared admission point, as that segment begins, before deciding whether native synthesis will play it. Transcript mode takes the same path and skips synthesis. It does not run a second formatter that happens to produce the expected fixture text while the GUI continues speaking broken fragments. Segments still waiting in the queue are not reported as heard; canceled future paragraphs cannot turn up later in the transcript as delivered speech.

This makes quiet operation useful without overstating it. The transcript is speech-channel evidence, not acoustic hearing. A record can carry the filtered text, its origin, and ordering; it cannot tell whether the speakers worked or a particular voice made an acronym intelligible. Native playback and transcript mode share admission, not the entire experience after admission.

The restriction on the listener is equally concrete. The accepted implementation described in the source is the separate public-API consumer at `examples/ch13-speech`, invoked by `tts-harness.sh`. It creates an Agent without builtins or skills

and registers exactly `hear`, `type`, `submit`, `wait`, and `sleep`. The older Node virtual-user program is history, not an additional current component.

`hear` returns new speech-channel entries and advances a cursor. `type` appends to the actual composer through browser input events. `submit` sends the owned draft and remembers which target turn belongs to the session. `wait` observes that turn's outcome, distinguishing success from failure or interruption; it does not return the answer. The listener still has to hear it. `sleep` provides a cancellable delay rather than another route to target state. Waits have a bounded timeout, with a 120-second default, and sleeps remain subject to session cancellation.

Privately, the helper uses Chrome's page-target protocol to drive the real client. Publicly, the listener gets no selector, click, evaluation, snapshot, shell, file, or skill-lifecycle tool. The consuming Agent's declarations matter as much as the peer's advertised surface: attaching a narrow peer to an ordinary coding Agent would leave other ways to peek. Dispatch also rejects unsupported operations and arguments, including guessed names. Asking a capable model to avoid the DOM is cooperation; removing the capability is confinement.

The helper settles its owned target before reporting success. The listener finishing its own model turn is not evidence that the target succeeded. Likewise, its final report needs to agree with both the observed speech and the actual target outcome. The report is an interpretation of evidence, not a replacement for it.

Pause Tracks Causes, Not Field Contents

Speech adds a second reason to hold automatic tool dispatch. A person may be listening to the current explanation, composing a correction, or doing both. For the local pacing policy, release follows only when neither cause remains.

Chapter 12's programmatic-input failure makes the representation important. The browser must remember human typing as a cause rather than repeatedly infer it from a nonempty field. MCP text does not become human text when an utterance ends. A genuinely entered space does count as composing, even though trimming the field would make it look empty.

The current browser described in the source centralizes the calculation in `updatePause(agent)`, with per-Agent typing and speech ownership and the Pause while speaking preference. Speech start, speech end, input, and submission update their causes through that calculation. It compares the derived value with the previous local value and sends a pause or unpause only on an edge. Extra keystrokes while already typing do not send more pauses.

Local causes govern automatic commands, not permanent ownership of the Agent's gate. An explicit unpause from another tab releases the Agent globally. An unchanged local cause in the first tab must not immediately reassert pause and veto that decision. This preserves the difference between automatic pacing and a person's deliberate override.

Escape cancels speech even when the composer contains text. It clears speech work without deleting the draft. Human typing may still hold the local cause, so stopping the voice and releasing the Agent are not synonymous. A check that sees only a typing pause and a later resume cannot establish this interaction. Speech ending first and typing ending first exercise different transitions.

Submission adds an ordering requirement. The browser clears the old typing hold **before** sending the new prompt. A late unpause can advance the continuation epoch after the prompt starts and invalidate its fresh speech. The stale cause must be released first; a delayed acknowledgement must also avoid clearing newer human typing. Otherwise the apparent speech defect is really a control message arriving in the wrong generation of work.

The Listener Has a Lifetime Too

The earlier Node peer closed its browser on stdin EOF but left a pending sleep timer alive. The Go client closed the transport and waited for the peer. Its model turn could be finished while cleanup continued waiting for a timer whose session no longer existed.

The source reports the revealing comparison: idle EOF exited in 125 milliseconds, while EOF during a 4000-millisecond sleep took 3865 milliseconds. That was delayed cleanup, not a permanent hang. Sharing the existing close-browser-and-exit handler between EOF and SIGTERM brought the corresponding pending-sleep probe to 127 milliseconds, with a 126-millisecond idle control. Normal sleep still worked while the session remained open. The fix ended the process that owned the timer; it did not need a new cancellation framework or an arbitrary sleep limit.

The current Go helper gives interactive stdin its own watcher. EOF cancels the context used by the listener, pending `wait`, and pending `sleep`, even while one is busy. Cleanup releases its draft only if it is unchanged and interrupts only a matching unfinished target turn. In normal launch mode the helper also closes the browser and GUI it owns. Attach mode disconnects

without closing the caller's processes. An ordinary one-shot invocation has no interactive stdin owner, so closed stdin does not cancel valid work.

Those ownership distinctions also shape attachment. The helper connects to the matching GUI page's debugger WebSocket, not the HTTP discovery endpoint or the browser-wide socket, and validates the expected GUI scheme, host, and path. An isolated save and browser profile keep an observation session away from someone's live work. The helper submits the required initial target prompt before starting the listener; the listener's goal is a separate instruction, not a substitute for starting that turn.

This remains browser-driving work in a small public consumer, outside the core. The GUI is itself a separate public consumer. Neither needs to absorb the listener's orchestration merely because the listener exercises its behavior.

Evidence at the Boundary That Can Fail

The source's automated application checks use an external provider and asynchronous synthesizer fake while retaining the real Agent, provider parser, renderer, and speech path. Observed utterance records carry text and source with sequence and timing; gate records describe actual pause and resume events and their causes. Constructing an expected log independently of the page would prove the log generator. It would leave the application unexamined.

Even an absence assertion needs a successful event behind it. The old file-suppression fixture requested `notes.txt`, but the harness launched the GUI in a different directory from the prepared workspace. The file read failed. Silence could not establish that successful file contents stayed out of speech, because those contents had never arrived. Using the planted file's

absolute path and checking the correlated successful result supplied something real to exclude. The model's distinct answer could then be compared with the transcript without confusing it with the file's bytes.

The historical speech checks passed before the blind client and cleanup correction were finished; the stronger file check later exposed a gap even in that passing speech result. Earlier client and command-lifetime behavior still needs separate compatibility evidence. Neither a successful speech score nor a listener process exiting zero establishes the whole system.

Native browser callbacks add evidence that the actual synthesizer started and ended utterances. The source carries Chapter 12's controls for whitespace, programmatic input, explicit other-tab unpause, and a subsequent real submission into this chapter. The earlier native check closed its extra control tab before the single-tab speech sequence; it did not resolve the unexplained two-tab queue stall. A quiet transcript check cannot fill that gap.

The source's three historical listener sessions illustrate the distinction between channel success and target-content success. All three used the restricted tools to type, submit, and receive real target transcripts. One target never chose the requested fruit, and its listener reported the omission; another supplied an apple in three sentences rather than the requested two; the third delivered thinking and response entries before the listener reported the apple. These were useful channel observations, not universal instruction following. No human listened to those runs. Separate native-callback checks and later deterministic cleanup evidence were not final-source repeats of the listener sessions.

The remaining human judgment is not a ceremonial final approval. Pronunciation, paragraph pauses, rate, and fatigue determine whether the channel is worth using. A browser accepting a rate value does not measure words per minute. The

source retains no measurement supporting its older fixed browser-speed ceiling or dedicated-engine comparison, so neither belongs in the design argument.

The listener tests dependence on speech; the transcript exposes delivered text and origin; independent target evidence exposes omissions; native callbacks expose playback transitions. A person still supplies the ears. Keyboard and screen-reader navigation remain separate accessibility concerns, and different blind and low-vision users need not want the same interface. Keeping this instrument narrow makes its findings useful without asking it to certify an experience it cannot have.

Chapter 14: The World's Best Context Engineering, Before Breakfast

Bill had the design in his head for a couple of months. One Tuesday morning he spent about twenty minutes describing it while CodeRhapsody turned it into a design record and checked it against the code. He took a shower; the first commit landed at 7:42, before breakfast. The ambition was to stop treating a long-running agent's memory as a transcript that eventually needs an indiscriminate summary.

The familiar symptom is an agent that starts rereading files and losing decisions. A new chat helps because a human chooses what to carry forward. The source chapter moves part of that judgment into the runtime: identify expendable tool traffic, remove it at explicit boundaries, and preserve the instructions and decisions needed to continue.

The event log records history. The context window is a working set rendered from that history. Removing bytes from the working set must not erase the record of their removal.

Give Every Entry a Removal Rule

The source reports that tool traffic dominated the measured sessions from Chapter 2: file bodies, command output, searches, stack traces. A file can be useful for one turn and wasteful on later

turns. The initial target is therefore tool traffic, not dialogue. This choice limits disruption, but it also limits the guarantee: the chapter does not bound the whole context window. Dialogue and retained instructions can still grow.

Each entry gains a kind identifying what can remove it. Dialogue contains ordinary conversation and tool traffic. Handoff contains a checkpoint written by the actor. Skill contains a loaded skill's rendered instructions. Tools contains changes to tool declarations. The last three are survivors: they never contain tool calls or results, so clearing tool parts cannot accidentally remove them.

This classification separates meaning from transport. A skill arrives through a tool call, but its instructions are not expendable merely because the load operation returned a result. Similarly, the checkpoint must outlive the call that created it. Shared vocabulary belongs in `internal/common`; History owns reduction of events into active context. Putting removal behavior on the common structs would give shared data a responsibility that belongs to the history owner.

Bill's earlier `compress_context` experience explains the caution. The model chose a message range and replaced it with its own summary. Bill reports that Gemini repeatedly cut from message one, where the user had stated the goal, and produced a poor summary. The model lost dialogue that existed in no file it could reread. The irreversible choice was not merely which output to forget, but which account of the task would remain.

The replacement, `handoff_task`, wrote a structured document and started a fresh instance with an empty window. That made the note solely responsible for continuity. It also added another path for durable state: CodeRhapsody's handoff path did not trigger the memory cascade used by `save_memory`,

a gap that remained in its notes for months. Ensemble builds neither tool. Its micro-handoff stays in the same conversation and uses the existing event path.

This design gives the actor a narrower job: decide what a completed piece of work has established. The runtime handles tool-byte removal through explicit rules. Visible dialogue, including the user's goal, remains intact.

Record the Cut, Not Just the Policy

Context management uses a target size, T, measured in bytes. Zero selects the default of 400,000; other targets are clamped to at least 20,000. T is a policy input rather than a hard cap on total context. It determines a result-stub threshold of T/100, a whole-tool-traffic budget of T/8, and a calls-and-stubs budget of T/16. At the default those are 4,000, 50,000, and 25,000 bytes.

Tool traffic ages in two stages. A `RedactResult` event replaces older result bodies with stubs but leaves their calls. A `RedactTool` event removes older calls and results together. A band can grow to twice its budget before a single event cuts an old prefix back toward the budget. That hysteresis avoids a stream of tiny cuts around one threshold and leaves a recent stretch of traffic intact. Under a prefix cache, changing an earlier byte invalidates the suffix after it. Cutting in steps preserves the useful prefix longer than shaving a few old bytes on every request.

Only completed batches already seen by the model are eligible. A result that has arrived but has not yet been included in a request is not expendable under this rule. Nor may a cut divide an unfinished call/result exchange. Batch completion and exposure to the model matter more than the age of an individual part.

The policy measures serialized tool parts, excluding dialogue and survivor bodies. Each event records its selected sequence boundary as a number. Replay applies that number; it does not recalculate the cut from today's settings. Otherwise changing T could change the apparent past: a restarted Agent might remove different content from the content removed during the original run. The retained event says what was cut, while settings explain how the live controller chose it.

The two removal stages preserve different amounts of continuity. A stub still tells the model that a call occurred and that its full output has been removed. Complete tool removal spends less context but loses that local trace. Neither cut deletes the underlying events. Their later availability depends on archive retention; a dialogue-only archive cannot reconstruct discarded raw results.

Addressless Stubs

On a model configured with `StubsToolResults`, results above the threshold become stubs in the request after the one that first carried them. The model gets one request with the full result and can use its next response to call `keep_tool_results`. That keeps the previous batch together. If the keep call travels alongside another file read, it does not keep that new read's result; its own acknowledgement is never stubbed. Keeping prevents this immediate per-round-trip replacement, not the later budget-based removal policy.

Other models do not receive automatic per-round-trip stubbing. The budget ladder still applies. In the source's feature matrix, the course Opus fixture enables both stubbing and inline

declarations, the course Sonnet fixture enables neither, and both flags remain false on existing real-model rows. A deterministic fixture is not evidence that a real provider supports its protocol.

Ensemble's stubs deliberately carry no address. CodeRhapsody supplied the counterexample: its stubs cited output files such as `cr/io/26`, but its handle counter restarted at one on every launch. After a restart, that path could name another command's output. The actor followed a plausible address, received plausible bytes, and had no indication they were wrong.

The first edition records CodeRhapsody's fix: keep the counter increasing across restarts. Ensemble makes the narrower promise that the bytes are gone from context and leaves recovery to the tools that produced them. A file can be read again, but it may have changed; rerunning a command can also have side effects. Neither operation guarantees retrieval of the original result. The original `ToolReturned` event preserves those bytes only while archive retention keeps it.

Checkpoint at the End of a Batch

`micro_handoff` lets the actor supply a short working note at a completed step: what succeeded, what failed, where the evidence lives, and what comes next. The note is the actor's selection of useful state, not a second model's summary of an arbitrary half-window.

The operation consists of three records: its tool call, an ordinary paired result, and a `MicroHandoff` event. The event's reduction waits until the batch's last result arrives. This matters even in a serial dispatcher. A tool beside the handoff may not have started when the handoff call returns; clearing immediately

would allow that later result to enter context after its call had disappeared. Siblings before or after the handoff must complete before the batch is cleared.

The committed event removes every tool call and result part, opaque reasoning, and private replay signatures, including signatures attached to visible text. Visible dialogue remains. Empty entries are dropped. Earlier Handoff entries leave the active context, and the new Handoff entry is appended, so only the latest checkpoint remains active and its text appears exactly once in the next request. Skill and Tools entries survive. Archived events and their sequence boundaries do not change.

Ordinary redaction is less aggressive. It preserves non-tool parts, including provider-private state; a handoff deliberately removes that hidden state while retaining visible conversation. Confusing these operations either leaves supposedly cleared private context behind or strips state that an ordinary tool cut was not authorized to remove.

The handoff event must be appended successfully before it replaces committed context. If that append fails, the earlier handoff and existing tool and reasoning state remain. The ordinary tool result is not, by itself, a durable checkpoint. Conversely, a later snapshot failure does not undo a handoff event already committed to the journal. This failure case is important because the happy path can look correct even when mutation precedes persistence: only the interrupted write reveals that recovery has no event explaining the in-memory cut.

Keeping every handoff active would rebuild the growth problem in another entry kind. The new note therefore carries unfinished work, decisions, and still-relevant commitments, not an accumulated history of completed tasks. Replacing older active notes does not itself remove their archived events. The checkpoint belongs at a completed micro-goal, not a token

threshold: clearing midway through a task can discard the result that still holds its unfinished state. Failed approaches deserve space in the note because otherwise the actor may repeat them.

Instructions Survive Their Load Calls

A `SkillLoaded` event records the rendered body and reduces to a Skill entry. Subsequent requests compose that body into SYSTEM using Chapter 9's instruction path, including after tool clearing and restart. Replay uses the recorded body rather than reopening a skill file whose contents may have changed.

The paired load result is not a second copy of the instructions. It is an acknowledgement on ordinary paths, or reference-only content on Anthropic's supported deferred-tool-search path. Mixing acknowledgement prose or a manual into that reference-only result changes the required wire shape. The instruction body remains owned by the Skill entry.

Unloading has a deliberately narrower effect than erasing instructions. It removes grants that no other active skill supplies and stops the skill's servers, but leaves its Skill entry contributing to SYSTEM. Ordinary clearing and this chapter's handoff do not remove it. A later verb owns that removal. A successor changing unload behavior must account for both capability and instruction lifetime; they are not the same state.

The initial primary instructions remain stable. Loading a skill changes SYSTEM and may invalidate a provider cache prefix. The design does not promise an unchanging cached prompt, nor clear history to conceal that change. Preserving instructions is more important than presenting a misleading cache claim.

Tool availability changes have their own event, `ToolsChanged`, carrying complete added declarations and removed names. Its `tool_decls` part is neither a call nor a

result. It therefore survives tool clearing without violating the rule that survivors contain no tool traffic.

Only the synthetic `InlineTools` course protocol freezes startup declarations and renders later changes inline in dialogue. Real models keep supported Chapter 9 disclosure paths: updated declarations or Anthropic's full deferred catalogue with references in genuinely paired successful load results. Remembering an inline declaration as history does not make it a valid provider request.

Clear Context Without Breaking the Next Request

A result can be correctly retained or removed and still leave an invalid wire message. The source's ordering failure makes that distinction concrete.

The first student reducer emitted a Tools entry after each individual result. The renderer treated that entry as a fresh user turn and placed declarations between two results from one call batch. An Anthropic-like ordering guard rejected the request. The required placement is after the complete user, tool-result, and ephemeral turn, not merely after the result that happened to trigger a declaration change. Serial tool execution does not remove the problem: the model issued one batch, and the renderer must preserve that request-level structure.

Calls and results form a unit even when their entries are separate. Clearing both sides of an exchange preserves that unit; retaining a result after removing its call does not. A declaration entry can survive cleanup without belonging between the parts of a tool exchange. The reducer's entry kinds and the renderer's message ordering solve different problems.

Replay and provider validity answer different questions. Replay can faithfully reproduce an invalid transcript. Inspecting the next request independently can establish pairing, declaration placement, and absence of hidden reasoning after handoff. An event-count assertion cannot establish those properties, and a test that uses the renderer's own helper to construct its expected request can reproduce the same mistake on both sides.

Keep Malformed Payloads From Partly Applying

History reduction is total over recognized events within a structurally valid stream. A known event with an unusable payload, such as a skill load without a body or a redaction naming absent entries, is diagnosed and skipped before mutation. Later valid events can still apply. A useful diagnostic identifies the skipped event and reason so the missing effect can be investigated. Skipping advances the replay position without changing context.

This deliberately changes Chapter 10's treatment of malformed known payloads. It does not relax JSON decoding, version, anchor, or sequence validation. Unknown event types still refuse the whole recovery candidate atomically, without installing an Agent or overwriting either recovery file. An implementation that cannot interpret an event cannot safely pretend that skipping it preserves meaning.

The distinction prevents two opposite failures. Panicking on every inapplicable known payload makes one bad operation prevent recovery of later valid work. Mutating partially and then announcing that an event was skipped leaves state with an effect that neither the reducer nor replay can explain. Applicability must be checked before any change.

Recovery Starts at the Backup's Boundary

A snapshot taken only at orderly shutdown cannot recover work completed after the last save when the process crashes. Chapter 14 requires complete events to reach an independent journal before publication or acknowledgement. Recovery can then reconstruct journaled progress after a process crash instead of depending on the last clean exit.

The journal's retention boundary is the validated backup snapshot, not the newest primary. In the source's example, the backup represents sequence 100, the primary 150, and the journal reaches 160. Keeping only events after 150 would leave no route from the backup to the newer work if the primary later proved corrupt. Recovery from that backup still needs 101–160.

Saving first encodes and validates the new state, then preserves Chapter 10's validated backup rotation and atomic primary replacement, and only afterward trims the journal. If no usable backup boundary is established, the required recovery tail must remain. A nominal backup filename is not enough; its validity determines whether it can justify deletion.

Recovery selects a valid primary or backup independently. It applies the selected snapshot's embedded tail, then applies only journal sequences beyond what the recovered state already covers. Overlap is expected and must not duplicate events. New sequence allocation continues beyond the recovered tail. Sequence boundaries make the merged history exact rather than a guess based on file age.

Append, save and rotation, and trimming must serialize at the actual event writer. A lock around an unrelated caller does not protect a concurrent append from being lost during replacement or trim. Likewise, failed append or save operations must preserve

the previously committed recovery and diagnostic bytes. Retry must neither duplicate an acknowledged event nor skip an unacknowledged one.

The promise is process-crash recovery. The source does not claim universal power-loss durability across operating systems, filesystems, and devices. A forced process exit before orderly save tests whether the journal carries recent work; it does not prove that every storage layer has made those bytes immune to power failure.

Diagnostic retention is separate from canonical recovery. Chapter 10's `log_mode` selects the archive policy and `diagnostic_max_records` limits diagnostic records; neither supplies a last-N cap on events needed to recover from the backup. After a successful handoff recovery save, the existing diagnostic writer performs its usual retention operation, the same one used at startup and exit. Its default is 200 whole records, zero means unlimited, and negative values are invalid; continued writes must still work afterward. Failed saves preserve both kinds of bytes. Creating a second truncation path would make retention depend on which lifecycle action happened to run.

One Policy Setting, Correct Ownership

The Context Management tab exposes `context_target`, not independent sliders for every derived threshold. One setting keeps the result threshold and the two budgets in proportion. It also avoids asking a person to predict the interaction of several removal policies before seeing a session run.

Settings belong to the owning Agent's data directory. The GUI uses `DataDir/settings.json`; the supplied application chooses `cr`, making its path `cr/settings.json` relative to its

launch directory. A similarly named file beside the executable is not interchangeable. The browser's public settings update reaches the owning Agent before subsequent policy work.

The browser must preserve unsent edits when a settings snapshot arrives. A server update is information about persisted state, not permission to overwrite a person's draft. The source's boundary checks tie editing to real requests: changing T must affect the next request without rewriting previously recorded redaction boundaries. The visible field, persisted value, live controller, and replayed past each have different responsibilities.

The architecture follows those owners. The Agent chooses policy from its configuration; History validates, journals, and reduces; Engine translates the result into vendor requests. Shared types do not acquire runtime behavior, and a new settings callback should not bypass the owning Agent. The GUI remains a separate public consumer, and the core retains the Chapter 1 client's behavior without taking a GUI dependency. A screen that can display the new setting does not establish that cleanup or recovery works, and a correct core reducer does not establish that the browser preserves an unsent edit.

Smaller Requests, Unsettled Model Behavior

A synthetic reference session in the first edition sent about 35 percent as many bytes with the ladder as without it, and about 13 percent with both the ladder and stubs. That measured the effect on scripted traffic, not real-provider cache behavior or answer quality. The stubbing run never triggered the ladder because it kept both bands below their budgets.

A historical `claude-opus-5` run illustrates the model's part in the policy. It read seven files, reread two, and never called the available `keep_tool_results`. The two rereads were the file it had not narrated and the largest file, which received only a short note. Files with fuller notes were not reread in that run. Dialogue preserved what the model wrote down; another tool round trip restored what its notes missed.

That observation supports a possible working pattern, not a general claim that one exposure is enough. The source explicitly leaves open whether stubbing interferes with model thinking. This run also reported no cache reads or writes: the reference renderer set no `cache_control` breakpoint. Preserving a stable prefix did not mean the application requested a cache. The run enabled features that this edition leaves false on real-model rows, so it is not a current deployment recipe.

The later multi-model observations show why correct cleanup is not enough. In a corrected-path Haiku 5.5 session, all six full file results arrived before redaction. The model subsequently claimed, "That file was redacted, so I never read it," and refused the handoff. The transcript disproved that account: cleanup removed already-presented bytes, not the original read. GPT-4o mini completed reads but repeatedly checkpointed until it reached the turn's batch limit. Of the five historical processes, all exited zero and only the Gemini session completed the requested task. Process success did not establish task success.

Those historical 5.5 names remain historical. Bill reports that 5.5 models do not work with ephemeral context and excludes them from supported use. The source names `claude-sonnet-4-6` as a supported live choice, without claiming that support guarantees successful completion.

Its bounded observation of the accepted Edition 3 build used that Sonnet model. Six human turns completed with full file results, six result cuts, and one tool cut. After handoff and

unload, the next captured request retained dialogue, SYSTEM manuals, and the handoff note but no tool traffic. The unloaded manual remained pending while `read_file` was no longer available. The observation guard then stopped on an unexpected or repeated handoff before the seventh final answer. The receipt remains failed: it establishes neither a completed post-handoff turn nor normal EOF nor live recovery. Passing local Chapter 14 and separate CLI Chapter 1 checks on the same held source did not turn this paid observation into a successful run.

The automated checks have their own limits. An early two-band test used paths too short to fill the calls band and therefore tested only result stubbing. Placing malformed events at the very end let a loader stop early without losing later valid work. Meaningful evidence must show both cuts actually occurring and valid events applying after a skipped event. Independent next-request inspection, disk-failure ordering, and browser edit behavior cover failures that a final model answer cannot reveal.

Dialogue still grows, even when tool cleanup works. The historical reading run's baseline request size rose as narration, calls, and stubs accumulated. Chapter 22 addresses dialogue removal. Until then, T remains a tool-traffic policy rather than a bound on the entire window, and reconstruction of earlier windows remains conditional on retaining enough canonical history.

Chapter 15: The Invisible Invoice

Bill reported $2,475 in model spend on the first edition through its recall chapter. He attributed a few hundred dollars to a caching bug. The answers were correct and arrived on time; the repeated input simply stopped receiving its discount. Nothing in the agent's behavior invited investigation.

Chapter 14 reduces the working set while trying to preserve stable prefixes. That does not establish that a provider caches them. Each request carries tools, instructions, and retained conversation; much of it repeats, but reusable bytes, correctly placed markers, and an actual cache grant are different facts. The source chapter connects them through three components: a lens that locates prefix changes, markers that identify reusable boundaries, and a meter that makes the resulting cost visible while someone can still act on it.

Observe the Request That Actually Leaves

The cache lens belongs to Engine and observes actual main sends, not every call to `Render`. Rendering can happen without a request being sent. Later compressor and judge conversations need their own accounting, but including them between main requests would make unrelated conversations look like prefix damage.

The lens rotates current and prior request files under the owning Agent's directory. The supplied clients use `cr/cache-current.txt` and `cr/cache-prior.txt`; startup resets these diagnostic files even when conversation history is restored. Named sections appear in the provider's cache concatenation order, not whichever order the JSON struct happens to emit. For Anthropic that means tools, system, then messages. A prefix is contiguous: comparing those sections in the wrong order gives the wrong boundary.

Each section is identical, appended, or changed. An array that gains elements at its tail is appended even though its closing bracket moves. Treating that as an edit would sound the alarm on every ordinary turn. Unexpected changes to stable sections deserve attention; deliberate skill, checkpoint, or model changes need interpretation rather than automatic diagnosis as defects. The files keep the evidence inspectable with ordinary `diff`, while structured verdicts make it usable by automation.

The conversation section is recognized by shape: an array of objects with `role` fields. Anthropic and OpenAI call it `messages`; Gemini calls it `contents`. An early lens exempted only `messages` from its stable-section alarm. Its Gemini report could therefore say `contents:appended` and simultaneously announce a broken prefix with a precise byte offset. The section comparison was right; the alarm interpreting it was wrong.

Markers must also disappear from the comparison. The history marker advances each turn, so yesterday's marked block is unmarked today. Those bytes are provider instructions, not changed conversation. Leaving them in the diff truncates the predicted reusable prefix at the old marker position.

Stripping them through a JSON unmarshal/remarshal cycle is tempting but changes key ordering and byte counts. Equality might remain consistent between the two transformed requests,

yet the measured bytes would no longer describe the request being diagnosed. The source instead strips the marker's shape in place while preserving every other byte.

The first-edition implementation got that shape wrong. It matched a constant containing exactly the compact marker emitted by the renderer, justified by the comment “we control the emitter.” But the lens saw the canonicalizer's `MarshalIndent` output, where spaces changed the marker's spelling. The strip matched nothing and failed without an error. Every compact, hand-built test passed. A replacement test using the actual indented shape exposed the defect: the retained review's two-turn comparison changed from an edited messages section to an appended one. The instrument needed a sample of the artifact, not another copy of its author's assumption.

A Late Reply Keeps Its Own Prediction

Engine's serialized turn decisions do not imply serialized HTTP completion. An interrupted worker can still finish after a new turn has begun. The lens therefore returns its predicted reusable fraction to the individual request, whose response callback carries that prediction back alongside the provider's usage.

Independent review demonstrated why a shared “latest prediction” is insufficient. It held an old cold response, interrupted the turn, and completed a new request. The predictions were 0.0000 for the old request and 0.8554 for the new one. Releasing the old response made it report 0.8554 too. Locking access to that shared value would remove a data race while preserving the attribution bug.

The corrected probe associated each response with its own prediction and counted both actual usages. A short lock protects comparisons and file rotation; none spans the HTTP wait.

Interrupting the conversation does not erase a charge already incurred.

This prediction is a byte fraction, whereas the provider's grant is a token fraction. Comparing the ratios can reveal a large discrepancy; adding or subtracting the raw quantities cannot. Tokenization is not linear in bytes, so close ratios are evidence consistent with reuse, not an identity or a billing audit. The original Anthropic report paired a 96.8 percent byte prediction with a 99.4 percent token grant.

Spend the Marker Budget on Different Boundaries

The source's explicit-cache layout spends a four-marker budget on distinct positions: the end of tools, the end of SYSTEM, the surviving compaction boundary, and the stable end of retained history. Each protects a different portion of the request. Missing content needs no placeholder just to spend a slot.

A marker on the last tool declaration banks the earliest prefix. The system marker normally includes those tools anyway, but cannot preserve them independently when the prompt changes. A separate tool marker lets unchanged schemas remain cached through a SYSTEM edit.

The system marker banks the instructions actually sent. Skill manuals belong in SYSTEM under Chapters 9 and 14, even when their presence changes the prefix. Moving a manual into a tool result to produce a better cache statistic would break instruction delivery. Likewise, editable memory is not promoted into frozen system instructions merely because stable bytes are cheaper.

History needs a rolling boundary. Marking the first user message adds only that message beyond the tools and instructions already covered by the system marker. Everything

learned afterward remains cold. Instead, the stable-end marker sits on the last eligible retained block and advances as history grows. Moving the marker does not invalidate the content it previously banked.

For this layout, a separate start-of-turn marker duplicates the rolling boundary from the previous turn. The providers can look backward through a bounded set of positions for a prior write; the client does not need to reproduce their exact lookback algorithm. Spending another slot on an anchor can obscure an actual prefix rewrite rather than cure it.

The stable endpoint is chosen after final tool-result adjacency has been established in the disposable wire view, without altering canonical history. It must stop before ephemera. Those delivered-once bytes are appended last and then cleared, sometimes sharing the final wire message with retained content. A marker after them banks a boundary guaranteed to disappear. An earlier prefix may still hit, which can make the placement mistake less obvious; the disappearing tail still cannot match. Capturing the retained boundary before appending ephemera avoids confusing message boundaries with content lifetime.

Compaction Changes What Can Be Banked

A `micro_handoff` removes tool traffic, opaque reasoning, and private replay signatures from the affected history. The old rolling marker may have banked bytes that no longer exist. The cache must rebuild from the retained projection, not from a reconstruction invented to preserve a discount.

The newest surviving handoff provides a useful compaction boundary. Its cuts were already recorded and applied by History; rendering does not recompute them. The inherited renderer projects the surviving note before transient ephemera rather than preserving a handoff dialogue message. Marker placement must

use a legal boundary in that actual projection. If none is available, the slot is omitted. Neither resurrecting old dialogue nor relocating the note into SYSTEM is an acceptable substitute.

A uniform handoff cut makes this boundary simpler than a policy that retains recent demonstration pairs. If “the three most recent pairs” is recomputed on every render, a newer handoff can displace pairs kept before an older one. The first surviving pair then becomes a moving boundary of its own. Ensemble's uniform strip avoids that extra dependency.

Chapter 22 adds a fallback to the last surviving memory entry when a memory fold removes the handoff. Rewriting memory near the front invalidates the suffix from that point, while earlier tools and instructions may remain reusable. That is a real cost of replacing memory, not a reason to keep obsolete content.

Similar Intent, Different Wire Contracts

Anthropic accepts `cache_control` on content blocks and tool definitions, allowing all four positions. SYSTEM must be an array of content blocks rather than a string.

The source's supported GPT-5.6 Sol Chat Completions route uses `prompt_cache_breakpoint` on message content blocks, with request-level `prompt_cache_options` selecting explicit mode. The tool array is not a message content block, so only three of the four positions are expressible. The system marker still covers tools during ordinary operation, but cannot preserve them independently through a prompt change. Support is selected from the model table; older rows retain implicit caching. These are model- and route-specific contracts, not interchangeable fields across OpenAI APIs.

Explicit mode has a particularly expensive empty case: it disables implicit caching. Selecting it without successfully placing any marker caches nothing. Anthropic's missing marker forfeits

an explicitly requested cache; OpenAI's empty explicit request also gives up the implicit cache it might otherwise have received. The opt-in follows evidence that at least one marker landed, not an intention to place one.

Gemini remains on `generateContent` with implicit caching. There are no markers to budget or separately managed cache objects in this design. Its eligibility threshold still matters. The source's October 11, 2026 documentation lists minima of 1,024 tokens for Sonnet-5 and GPT-5.6 Sol, and 4,096 for Gemini 3.8 Flash. Those dated values require the selected model and route; request-file bytes are not a substitute for tokens. A small prompt with no cache reads has not demonstrated a cache defect.

The Provider Must Actually Grant the Read

Historical live runs show why plausible requests and correct answers are insufficient evidence. They also need their experimental conditions attached.

The first-edition Anthropic run reported cache reads of 0, 3,627, and 3,648 tokens. The 21-token increase matched the previous exchange, showing that the advancing marker banked more history in that run. It does not establish a fixed increment for other responses or providers, and the measurement predates the final marker arrangement.

An OpenAI experiment initially removed a marker from a primed position and got zero reads. It also changed the message from an array of content parts to a plain string. The apparent explanation, that removing the marker had broken the prefix, had not been isolated. A second probe changed only the encoding and received 7,813 cached tokens out of 7,816. That tested prefix

survived the encoding change; the missing lookup breakpoint explained the earlier zero. A probe changing two variables had offered a much more confident diagnosis than it earned.

Gemini's observations depended on conversation shape. The original small-prompt script produced three zeroes, initially diagnosed as unimplemented caching. An eligible run corrected that interpretation. Later probes used a fixed large user message, a large stable system reference with small growing history, and history growing by roughly 8,400 tokens per turn. The first two warmed on turn two; the growing-history run began with two zeroes. An early review tried to challenge the history result using the fixed-message probe. Both measurements were real, but they answered different questions.

A later browser run with a large stable reference also warmed Gemini on the second request. Its changing read counts establish neither a universal 4,096-token block rule nor a rule that partial blocks remain uncached forever. Stable content belongs early, but “stable for this demonstration” does not authorize placing arbitrary retrieved text in instructions. Warm-up and the amount granted must be observed on the shape the application actually sends.

Fake providers remain valuable for request structure, parser behavior, and accounting. They grant whatever counts the fixture supplies, including below a real provider's minimum. A larger byte fixture does not fix that evidence boundary. The source's numeric marker fixture also has no handoff or ephemeral tail, so it cannot establish those placement rules. Focused observations cover browser rendering, interrupted-worker attribution, and compaction boundaries separately.

Real use can reveal a contract the fake never modeled. The historical GPT-5.6 Sol Chat Completions run refused function-tool requests with reasoning effort. Omitting `reasoning_effort` did not repair it: omission selected the

provider's reasoning default and produced the same refusal. That route needed the explicit value `"none"`. A faithful regression can preserve the discovered restriction, but a fake cannot discover an absent contract by itself.

Count Once, Price at Display Time

Usage records four disjoint categories: ordinary input, cache writes, cache reads, and output. Prices live on `ModelFeatures` as full dollar rates per million tokens. A cache-write rate is not a surcharge to add on top of ordinary input. Each token belongs to its applicable bucket and is charged once.

Vendor responses do not arrive in that shared form. Anthropic reports disjoint ordinary, write, and read input. OpenAI's prompt total already includes cached reads and writes, so those buckets must be subtracted to derive ordinary input. Gemini's prompt total includes cache reads; its charged output includes reported thinking. Streams retain final usage rather than summing cumulative updates.

Within a valid usage object, an omitted optional cache bucket means zero. Missing usage, missing required input or output counts, and invalid supplied counts remain unknown. Treating any of those as free usage makes an accounting failure look like a bargain. An unknown price likewise displays a dash, while available token counts remain visible.

Each request's counts retain the model and applicable price tier that incurred them. Engine derives cost from those identities and rates when rendering the meter, rather than storing dollars in the event log. A price correction can then change the derived cost without rewriting measured history; switching models cannot reprice earlier work as if the new model had performed it.

The first implementation's meter looked up the model named in settings. When the environment selected the model, that setting was empty. The lookup failed and a paid Sonnet run reported `priced:false, cost:0`. The instrument intended to expose waste was concealing it. The correction takes the model actually in force from Engine configuration at the composition root. The retained review attributes this finding to the grader, not to the live run: ordinary application ownership was enough to explain and reproduce it.

The Meter Has Two Time Scales

The GUI header displays session cost, last-response input tokens, last-response cache-read tokens, and session cache hit rate. “Input” here means all input to the latest main response: ordinary input plus cache writes plus cache reads, each once. The public `usage` frame exposes those disjoint buckets separately; the header combines them. A tooltip holds session totals so the current prompt size is not buried under a number that can only grow.

Engine owns process-lifetime accounting and separately retains the latest main-response counts. Neither is restored from `save.json`. Conversation history can return after restart while the meter starts at zero, because no response has yet been charged to the new process. Restored historical usage answers a different question from the cost since the developer sat down.

Chapters 22 and 23 add compressor and judge calls to session totals and cost, priced by the model that performed them. They do not replace the latest main-response display and do not enter the main-request lens comparison. The distinction keeps a small auxiliary response from masquerading as the size of the actor's current prompt while still charging it to the session.

Earlier October 11 browser sessions on Haiku 5.5, GPT-5.6 Sol, and Gemini 3.8 Flash each returned four correct answers, with provider usage independently compared against visible meters. Their session hit rates included the cold first request, unlike a last-response rate. Haiku 5.5 remains a historical identity, not a current supported choice. Those captures had no restart or handoff and preceded the lens-overlap correction, so they could not establish those behaviors.

The unchanged original grader, meanwhile, returned 0/100 because its primary skill fixture omitted the required type and the GUI never started. That was a setup failure, not evidence of seven broken cache features.

The source's accepted October 22 Sonnet 4.6 run returned four correct answers. Provider usage yielded $0.0755091; the header displayed $0.0755 and the tooltip $0.075509. Restart replayed the conversation with a zero process meter.

Chapter 16: Leaving Anthropic for OpenAI

A provider migration tests whether the agent owns its conversation or merely keeps a copy of one provider's request. Ensemble already has a neutral history, provider renderers, and a transport boundary. Moving to OpenAI still requires a new renderer, different stream parsing, and credentials that can renew themselves. It should not require rebuilding the Agent or teaching every tool how authentication works.

The reason for moving is a conflict over context control. The agent deliberately removes stale material: ephemeral views, old tool output, and context selected for compaction. A provider may legitimately protect hidden reasoning while imposing rules that make those edits incompatible with retaining its reasoning artifacts. The choice is not between security and no security. It is whether the service's continuation contract fits the agent's context policy.

Hidden reasoning has an authorization boundary

Encryption does not by itself establish who may replay a reasoning artifact. Its owner may be unable to read it while another model can decode it. In August 2026, researchers replayed a frontier model's encrypted reasoning into a cheaper

model behind a jailbreak prompt. The receiving model treated the block as its own prior thinking. The recovered material included credentials; 64 of the 704 recovered artifacts were absent from visible chat history. An apparently opaque conversation attachment had carried secrets outside the visible record. These are the researchers' findings, not credentials to try. (Panfilov et al., §§3.1 and 4.1.)

OpenAI's September 30 report describes stronger protections across users, workspaces, organizations, and model families, a closed replay pathway, and output checks. It does not disclose enough implementation detail to identify the mitigation as exactly the paper's proposed cryptographic design. Protecting replay means binding an artifact to an authorized context, not merely making its bytes unreadable to the person storing them.

Anthropic's preserved-thinking policy creates a different constraint. The Opus 5.5 launch applied it to that model and Fable 5.1 for API accounts created on or after August 31, 2026. Documentation checked October 11 ties retained thinking to the prefix against which it was produced: editing earlier messages or tool results can invalidate later thinking. It also supports server-side editing and client-side compaction that discards stale thinking. The conflict is therefore not "compaction is forbidden." It is retaining those artifacts while independently changing their earlier context.

Ensemble depends on that editing authority. A changing screen view, an ephemeral tool result, and a recall attachment retained until a checkpoint have different lifetimes, but the client must be able to stop sending each when its policy calls for it. Current Anthropic examples elsewhere in the edition stay on supported older models such as Sonnet 5; a newer catalog entry does not establish compatibility with that behavior. This chapter takes the migration route rather than redesigning context management around preserved thinking.

Credentials belong beside transport, not inside rendering

A static API key is constant from this client's perspective: resolve it at startup and send the same value with each request. The service can still expire or revoke it. An OAuth access token, by contrast, expires routinely, and renewing it rotates the refresh token. Treating either token as an ordinary configuration string puts renewal and concurrency failures into every caller.

The credential seam has two operations. `GetBearerToken` returns a valid bearer or an error; `Kind` distinguishes an API key from OAuth. The API-key provider wraps a string. The OAuth provider owns protected profile state and coordinates refresh. HTTP obtains the bearer just before sending and puts it in the same `Authorization` header in either case. After an authentication failure, HTTP can request renewal without taking ownership of stored tokens.

The owner relationships matter as much as the interface. Ensemble owns the auth store. Agent selects the funding mode. Engine obtains the bearer through request-local configuration. The renderer receives only public route capabilities, such as whether tools require a namespace. Turning history into request bytes must not require access to an account or trigger a refresh. That keeps saved-conversation rendering and replay offline, and lets a new renderer concentrate on its wire format rather than reimplementing OAuth.

Signing in without a secret hidden in the binary

A distributed local application cannot keep a shared client secret: someone can inspect or copy the binary. Authorization Code with PKCE instead proves that the application completing authorization is the one that began it.

The client generates 32 cryptographically random bytes and base64url-encodes them without padding to make a verifier. It hashes the verifier string with SHA-256 and base64url-encodes the digest to make a challenge. The authorization request carries the challenge; the token exchange carries the verifier. Intercepting the authorization code is insufficient without the verifier.

The client briefly becomes a web server, binding a loopback listener on any available port at `127.0.0.1`, with the exact path `/auth/callback`. For this contract, `localhost` is not a substitute and `/callback` is not an equivalent path. Only the port varies. The browser returns the authorization result there, and the listener shuts down after receiving the callback.

PKCE does not replace the other checks. Generate fresh `state` and `nonce` values for each authorization attempt. The state must match the callback to prevent cross-site request forgery; the nonce must match the returned ID token to prevent replay. Persist a random `urn:uuid:<v4>` as the installation's `ext_agent_host_id`, never deriving it from username, hostname, email, or MAC address. The RFC 8707 resource value, `https://api.openai.com/v1`, appears in both the authorization request and the code exchange.

First registration uses the bootstrap client ID `dynamic_agent_client`. The callback supplies the issued `client_id` alongside the code and state. Persist that issued ID

before exchanging the code: the bootstrap value is never valid for token exchange. Subsequent requests use the issued ID. A returning callback may omit it, but a different supplied ID is an error, not a new identity to accept silently. This is callback-based registration, not a separate RFC 7591 registration request. Send `agent_name_hint=Ensemble` on first registration and omit it on reauthorization. Identity scopes, offline access, `resource.invoke`, and `chatgpt.tokens.use.direct` are protocol inputs rather than descriptive labels. (Sign-in contract.)

Identity and permission to spend are separate results

The exchange returns an access token for API calls, a refresh token for renewal, and an ID token for identity. Decode-and-trust is not ID-token validation. Fetch the trusted issuer's published JWKS, verify the JWT signature, and check that the issuer is exactly `https://auth.openai.com`, the audience is the issued client ID, the token is unexpired, and the nonce matches the authorization request. The `sub` claim supplies stable account identity. Keep it with the issued client ID; the same email address or person in a different workspace registration is not grounds for merging token sets. A returning verified identity must match the account being renewed before replacing its credentials.

A valid identity still does not authorize plan inference. Inspect the scopes actually granted in the token response for `chatgpt.tokens.use.direct`. Without it, identity may be saved, but inference remains disabled. Trying an available API key instead would turn missing permission into an unauthorized funding change.

The discovery document inspected during the original implementation advertised identity and offline-access scopes but not the plan-usage scope. Implementing only what that catalog listed could therefore produce a working login that could not infer. Discovery describes available protocol metadata; the granted scopes determine what this user authorized.

Refresh is a coordinated state change

The documented lifetimes on October 11 were about an hour for an access token and thirty days for a refresh token. Those are not expiry constants to bake into the client. Track the returned expiry and refresh before the next inference when the current time reaches five minutes before `expires_at`.

Refresh rotation creates the dangerous race. Once the server issues a replacement refresh token, the old one is dead. If two callers independently notice expiry, the first can renew successfully while the second submits the old token. The server may treat that second attempt as replay and invalidate both tokens. A mutex protects goroutines in one process; it cannot protect two clients sharing a credential file.

The accepted store serializes refresh per profile across processes, then reloads the profile under that coordination before deciding to renew. The second caller can use the replacement already saved by the first. Save the new access and refresh tokens atomically before using either, through a transaction or temporary-file rename. Updating fields separately risks leaving a mixed, unusable profile.

Atomic replacement solves partial local writes, not the entire distributed transaction. If the server rotates the token and the client crashes before persisting the response, the old token is

already invalid and the new one is lost. No local rename can recover a response that never reached durable storage.

Terminal refresh errors are `invalid_grant`, `invalid_refresh_token`, `token_expired`, `refresh_token_expired`, `refresh_token_invalidated`, and `refresh_token_reused`. They establish that the credentials are unusable. Discard the tokens, retain the issued client ID and host ID, and require sign-in again. Network failures and HTTP 5xx are transient: retain credentials and retry with exponential backoff. A reachable server can have a temporary failure too.

An inference 401 alone does not prove that refresh credentials are revoked. This implementation attempts renewal once through the selected funding source. Only a terminal refresh rejection warrants clearing the tokens. Authentication recovery must not become a hidden switch to another account or another bill.

Responses changes encoding and the meaning of completion

Responses represents the same neutral conversation with input items rather than the previous message format. System-role messages become developer-role messages, and streaming event names change. HTTP 200 means the transport opened. `response.completed` establishes completed inference. Text appearing on the connection is not enough to make a failed or incomplete response part of the conversation.

The model table selects the surface per model through `SurfaceResponses`. Earlier models remain on Chat Completions; selected newer models use Responses. A boundary test guards against flipping an entire vendor onto a surface that

has not been established for every model. Historical preparation for this migration included an unwired Responses surface, separate reasoning-summary events, and recognition that reasoning with function tools needed Responses. Those were useful seams, not an already-completed migration.

Model support and route permission are different facts. On the ChatGPT-plan HTTP route, `store: false` and `stream: true` are mandatory and `input` carries the required history. The route excludes sampling controls `temperature` and `top_p`; `max_output_tokens`; server-side continuation fields `previous_response_id` and `conversation`; `background`; and `metadata`, `max_tool_calls`, `moderation`, `multi_agent`, `prompt`, `prompt_cache_retention`, `safety_identifier`, `top_logprobs`, `truncation`, and `user`. Keep these restrictions in the selected route profile, not scattered through the message loop or inferred from the model name.

Local function tools need a namespace on this route. Preserve the returned namespace separately from the function name, then carry the call ID into `function_call_output`. Producing a tool call is only the first half of the transaction. The next request must contain the actual tool result correlated with that call.

Cache support follows the route

The top-level `instructions` field is legal on the plan route. As a plain string it cannot carry its own content-block breakpoint; that does not make its bytes inherently uncacheable, because a later reusable prefix can include earlier instructions. This renderer places the constitution in a developer-role content block so the supported API-key route can put an explicit boundary there.

The earlier student run exposed a restriction missing from the original account. Its first subscription request failed with HTTP 400 and `invalid_parameter`, naming `prompt_cache_options`. Explicit mode had worked with the API key. The accepted plan profile therefore omits that option and its associated breakpoint markers. The option's rejection was observed; omitting the paired markers is a conservative choice, not a separately measured rejection.

The supported API-key route retains `prompt_cache_options.mode` set to `explicit` and markers on eligible content blocks within the four-marker budget. GPT-6.1 Sol standard prices verified October 11, for prompts through 272K input tokens, were $2 per million fresh input tokens, $0.10 cached input, $2.50 cache writes, and $10 output. Reads cost 0.05 times fresh input and writes 1.25 times. Those prices explain why placement matters on that route. They are neither a subscription invoice nor evidence that any particular request received cached usage.

A completed tool item can precede an empty terminal output

Removing the unsupported cache option did not make the subscription run work immediately. The body contained SSE, but the parser chose whole-JSON decoding. Once stream selection was repaired, another assumption failed: `response.output_item.done` had already delivered a completed `read_file` item, while `response.completed` carried an empty `output` array.

The parser must detect streaming without buffering the whole body, retain completed items by output index, and recover them when terminal output is empty. A nonempty terminal output

remains authoritative. In both cases, terminal success and usage must arrive before retained items become conversation facts or executable calls. Complete-looking arguments do not authorize executing a response that later fails.

The captured stream reproduced both defects without another model call. A fake that always repeats completed items in its terminal array cannot reveal the lost-item bug. The real route was less redundant than the test fixture, and the parser had mistaken redundancy for a requirement.

Public summaries need their own live path

Reasoning summaries are model-generated descriptions of private thinking, not the private trace. Preserve encrypted reasoning for compatible replay, but do not read or speak its ciphertext. Public-summary deltas belong in their own observer event kind, separate from assistant-text deltas, with existing part identities associating each stream with its final content. Interleaved reasoning, tools, and assistant text must retain stream order without losing summaries next to tool calls.

Request `reasoning.summary: "auto"` by default on supported models; an explicit supported choice such as `detailed` must not silently become every request's setting. A model may also require ordinary assistant explanation before tools. Public summaries do not satisfy that rule. Select it from the actual model configuration, including a worker's model override, put it in model-facing instructions, and retain the dispatch and arrival-order checks. Whether a summary happened to arrive is not a capability signal.

This is an accessibility requirement, not just an extra panel. Bill has no macular vision and reads code at about 80 words per minute; he listens to model output at about five times speaking rate. A summary delivered after the answer misses the opportunity to hear a wrong approach developing and send a corrective hint. The GUI needs phrases while generation continues, without inserting those phrases into the assistant's answer.

The first-edition coder investigated streaming before implementation. The October 2 report recorded 314 deltas across three parts for `gpt-6.1-sol` with high effort and a `detailed` summary, 186 with `auto`, and four with `concise`. An initial trivial prompt had produced only 18 reasoning tokens and no summary deltas, nearly leading to a false capability conclusion. A probe needs a task that exercises the capability it is meant to observe.

Even substantial reasoning does not guarantee a public summary. The earlier student API-key session produced 90 summary deltas. Its subscription session requested `detailed` and used 165 reasoning tokens on the final response, yet produced none. Completion and shutdown cannot depend on hearing a summary.

The Edition 3 restart found a separate delivery gap: final summary cards survived, but the live path missed text supplied only in the final summary. Sending the unseen suffix through the existing public-summary path makes it available to display and speech without repeating the prefix. A complete card proves neither incremental arrival nor native speech delivery; those are separate boundaries.

Funding selection must survive failure

The proposed billing state machine had three modes: subscription-only `chatgpt`, API-key-only `api`, and `chatgpt_then_api`. The third would require explicit `allow_metered_fallback`, defaulting to false, and an announcement before the first metered request. A paused agent is preferable to a bill the user did not authorize.

Only `subscription_sharing_usage_limit_exceeded` permits that proposed quota-driven transition. The limit can be app-specific rather than exhaustion of the whole subscription. `subscription_sharing_usage_unavailable` means temporary overload and calls for backoff, not another funding source. `subscription_sharing_unsupported_capability` requires request adaptation or a reported limitation. A 401 calls for credential and permission diagnosis with bounded same-source renewal. Treating these failures as interchangeable would make billing depend on unrelated transport or feature errors.

The accepted implementation chooses the smaller contract: it rejects `chatgpt_then_api` and provides no automatic fallback. Both supported modes announce their selected funding source before the first request. Successful login selects `chatgpt` for later launches unless explicitly overridden. Logout leaves that selection in place. Removing subscription credentials must not silently activate an API key already present in the environment.

Model discovery is account-specific too. `GET /v1/models` with OAuth returns the subscription catalog, filtered by `visibility: "list"`; use `slug` in requests and `display_name` in the UI. An API key sees its tier's catalog instead. Availability there is necessary but not sufficient: the application still needs a supported model route. This integration

uses the public `https://api.openai.com/v1/responses` endpoint, not another application's private credential store or backend.

Login, status, discovery, inference, and logout must use the same profile selection. The client uses `EN_HOME` when supplied, otherwise its own `ensemble` directory under the operating system's user-config location. Access tokens, refresh tokens, ID tokens, and API keys belong only in protected credential storage, never in prompts, memories, conversation saves, logs, debug output, or other model-visible state. Printing an authorization URL is not permission to print tokens. Logout attempts remote revocation, removes local tokens, and retains registration and host identity. If remote revocation cannot be confirmed, report that uncertainty rather than claiming it succeeded.

What the recorded runs establish

The local exercise drives actual clients against external OAuth and Responses fakes. Signed JWTs, published fake JWKS, rotating tokens, and actual tool-result bytes test more than whether a request-shaped object can be constructed. A login that never reaches inference cannot establish a billing rule. Copying an error code into output does not establish that Agent treated the turn as failed. Both GUI and CLI login entrypoints should delegate to the same auth owner rather than acquire independent implementations.

The synthetic `gpt-ch19-course` model declares its fixture capabilities explicitly: Responses, SSE, summaries, encrypted continuation, and API-key explicit caching. It is not a live model, and its name is no basis for inferring capabilities of other OpenAI models. Local issuer, inference endpoint, and private-profile

overrides keep these exercises separate from real browser consent and production accounts. Multi-turn continuation includes encrypted reasoning rather than testing only an isolated answer.

A local score samples the contract. Owner relationships, identity-validation cases, cross-process refresh coordination, and atomic replacement still need review. Browser checks of incremental speech add evidence that visible WebSocket events alone cannot supply. Neither fake summaries nor a passing score guarantees that a live model will emit a summary.

The historical paid task required reading a scheduling file, finding a minimum schedule for two identical workers, and proving optimality. The dependency chain A–C–F–H gives a lower bound of 13 from durations 4, 5, 3, and 1. That chain fits on one worker while the other performs B from 0–3, E from 3–7, D from 7–9, and G from 9–11. Matching a feasible schedule to the lower bound establishes optimality. Knowing that answer beforehand does not establish tool use: the actual file result must reach the following request.

The earlier API-key run did read the file and produce that schedule. Its three requests included the original prompt and a hint-triggered continuation, not two independent prompted turns. The hint reached the third request after an intermediate answer, and the process exited normally in 14.51 seconds.

The corrected subscription run made two requests with no API key configured. It read the file, carried the correlated result into the second request, and completed the same optimal answer in 14.887 seconds. Both terminal output arrays were empty, directly exercising completed-item recovery. But the launcher had tied its hint and eventual EOF to seeing a public summary. None arrived. The answer finished while the idle CLI remained open until the launcher killed it at 150.03 seconds. That is evidence of a successful answer and tool loop, not graceful

shutdown or a saved conversation. The corrected completion condition was checked against captured events; no new paid run was made.

The cache lens also needed repair. Its extractor recognized the older API's messages but omitted Responses `input`, so it compared the tools while ignoring the conversation. After including the actual input, the final plan request pair shared a predicted 92.12% byte prefix, yet both provider cache counters remained zero. Each input was below 1,024 tokens. Stable bytes, a completed task, and a granted cache hit remain distinct observations.

On the October 22, 2026 Edition 3 restart, local chapter and regression checks passed. Bill deferred the live OAuth exercise. There was no fresh native GUI, browser, or listening observation accompanying those results. The earlier paid runs remain historical evidence, not proof that the restarted implementation worked with a live account or provider.

Chapter 17: The Crossover

Bill's first crossover was supposed to take a morning: replace CodeRhapsody, the agent used to write the book, with Ensemble, the agent built from it. The first-edition account records twelve days, thirty-seven changes, and a provider caching bug. Passing the existing checks had not prepared the new agent for an ordinary workday.

The failures were not all spectacular. Some answered every request correctly while using the wrong model or credential. Others saved the entire conversation and displayed none of it after restart. The cache diagnostic was particularly industrious: it reported a problem on every request, making the real problem indistinguishable from normal operation.

These are historical incidents from the first-edition crossover changelog, begun September 27, 2026, not a defect list for the current runtime. Chapter 16 supplies the implementation predecessor for this edition. This chapter adds no implementation and reports no fresh crossover or private-memory migration. Its later October 11 coding run is a separate, second-edition observation.

The visible half can work alone

The second assistant response appeared inside the first response's bubble. Each vendor parser started its part counter again for each API response, so every turn could contain a

`part_id` of 1. The GUI retained its artifact map across turns. New text therefore found an old DOM node and appended there.

Clearing the map at each turn would have hidden the collision, but made the client responsible for uniqueness and dependent on receiving a turn-boundary signal before the first delta. The historical repair put a single monotonic counter on the actor, shared by parsers and both delta and final emitters. A content address then remained unique throughout the session.

The settings dialog presented a subtler illusion. Every control called `sendPatch`, which had never been defined. Theme and font size still changed locally, so the operator saw the effect while the persistence call failed. Four wrong hypotheses preceded the harness log, whose first line named the cause. Temperature, maximum-token, and system-prompt controls did nothing at all; they were removed rather than left to advertise nonexistent behavior.

Even the working local font adjustment had limits. Absolute pixel sizes ignored a change to the root font size. Converting the stylesheet to `rem` made the controls scale together. The replacement multiline input also needed its height cap in relative units; a fixed pixel cap would hold fewer lines as the text grew. A zero stored font size meant “use the default,” not a meaningful size to apply.

Shutdown failed for a different reason. The CLI saved after its blocking stdin loop returned, but Ctrl-C terminated the process before that path could run. There was no race to win. A signal handler that saved and called `os.Exit` would still skip load-bearing deferred cleanup. Moving stdin reading into a goroutine let the main goroutine wait for either input completion or a signal, then leave through the existing cleanup path.

Daily use exposed these gaps quickly. Once understood, they were ordinary regression targets: another turn, a settings reload, a signal, larger text. The missing evidence was not whether the

application could display a response, but whether its connected behaviors survived the next action.

A readout must be able to disagree

The model dropdown wrote one configuration field while startup read another. Requests continued to use the startup default. Worse, `effectiveModel` preferred the settings store because that reflected what the operator had just selected. The readout showed the intended model, not the model making requests.

A readout fed from intent always agrees with the operator. A readout fed from execution can reveal that the selection failed.

Changing providers exposed the same gap in funding. The historical runtime switch already moved model, vendor, API surface, base URL, and API key together. The credential provider was introduced later and attached only at startup, behind a vendor check. Once models became changeable during a session, that startup check no longer established which credential belonged on the next request.

Switching from Claude to GPT changed the dialect and destination without bringing the plan credential along. A metered key could answer while bypassing the intended subscription. The changelog inferred that fallback from code and a 401 response; it did not observe a charge on an invoice. The risk was real without the invoice claim: an apparently successful answer could conceal a funding mismatch.

The repair added the credential provider to the state already moved by the switch. It also made the effective credential observable. `DescribeCredential` could name the credential kind but had no path to emit a token. That placed secrecy in one owner rather than requiring every logging call to remember

redaction. An absent provider was described as a static key, not "none," because the former named the funding behavior that mattered.

Startup ordering caused another wrong-model session. The resolver selected a vendor before resolving the stored model, so an Astra selection could produce a healthy-looking session running Sonnet. The diagnosis concerned stored choices outside the environment model override. The browser happened to send its selection after connecting, correcting the mistake; the CLI never did and stayed wrong throughout.

The repair resolved the model first and derived the vendor from its table entry. An explicit vendor override still took precedence, allowing a local OpenAI-compatible endpoint to serve an identifier unknown to the table. This is not a reason to eliminate overrides. It is a reason to make the default dependency run in the right direction.

One conversation, two delivery paths

Live streaming and replay shared a WebSocket protocol, but not enough of their construction logic. Restarting exposed the difference.

The hub's renderable-history count advanced only when it observed live events. Restoring a full log left that count at zero, so replay iterated over nothing. Initializing the count from the supplied log exposed the next failure: sends silently dropped messages when a buffer filled. A long conversation exceeded the historical 256-slot queue.

A newer progress snapshot may replace an older one. Text deltas, tool results, and replayed artifacts cannot be discarded on that assumption. The repair waited for capacity, with a timeout to

avoid retaining a dead connection indefinitely. The durable requirement is preserving content or terminating the failed connection honestly, not preserving that particular queue size.

Another restart showed user messages and tool calls but no assistant text or thinking blocks. All 392 events were in the save. The replay conversion omitted `part_id`, so the GUI placed responses under the same undefined identity. The live conversion supplied it. Adding the missing field repaired the immediate symptom; sharing content identity across both paths addressed the underlying design mistake.

Reset failed in the opposite direction. The saved conversation cleared correctly, but the screen kept its old text until refresh. Recording an event appended it to the log and applied the reducer; it did not notify the observer. A live `ConversationCleared` observation made the reset visible without relying on replay to repair the screen later. Inspecting the saved record separately from the live view located the failure. Refresh alone would have concealed it.

The browser also tried to decide whether input was a new prompt or a hint by consulting its copy of turn state. After a mid-turn kill left stale in-flight state, it sent hints to a turn that no longer existed. The connection stayed open while messages disappeared.

The repair removed that decision from the client. It always sent a prompt; the actor classified the input by where it received it. At the main loop the actor was idle. At the mid-turn drain it was running a turn, so the input was a hint. The control flow supplied the fact that the browser's copied state had failed to preserve.

The CLI initially bypassed this actor and called a bare engine loop. It could not reproduce the GUI's failures because it was effectively exercising another agent. Rebuilding it as a text observer over the same actor made the comparison useful. A

failure in both renderers points toward the shared path; a failure in only one points toward its input, timing, or rendering. Neither observation alone proves the core innocent.

The diagnostic that cried divergence

The crossover record reports a morning consuming 46% of a weekly plan allowance: 175 requests, 6.9 million uncached input tokens, and a 22.5% hit rate. Those are recorded usage and allowance observations, not an independently reconstructed invoice. The answers still arrived normally.

The cache lens was meant to explain such costs. It split request bodies into named sections in the provider's actual cache order and compared them with prior requests. Stable local bytes, a changed prefix, and a provider cache miss despite stable bytes are different findings. The last leaves routing, eligibility, expiry, or eviction to investigate; local comparison cannot identify which occurred.

The lens first had to learn what growth looked like. Appending to a JSON array moves its closing bracket, so a naive string comparison calls every append a rewrite. Comparing sections after trimming the prior container's trailing framing distinguished an unchanged prefix from ordinary growth.

A second mistake defeated that distinction on Responses. The canonicalizer recognized Chat Completions' `messages` and Gemini's `contents`, but not Responses' `input`. It treated the conversation as a frozen section. Every normal appended turn triggered the same divergence warning. A genuine rewrite that reduced the surviving prefix to 34% looked exactly like the harmless appends around it.

Reading the conversation key from the body repaired that classification. Reporting growth separately from rewriting, with the offset where matching stopped, made the warning useful. The problem was not absent instrumentation. The earlier cache chapter had already supplied a lens and meter. The instrument was observing the wrong thing with great consistency.

Breakpoint placement needed the same scrutiny. One marker at the previous exchange repeated a boundary already offered by the earlier rolling marker. In the tested explicit-cache layout, its slot was more useful at the end of the tool array. Tools came before the system prompt in cache order, so that boundary could preserve an earlier eligible prefix when the prompt changed.

The resulting possible boundaries were tools, system prompt, retained checkpoint, and the last eligible retained content before ephemera. If a handoff had been removed, surviving memory could represent the checkpoint. That memory remained data, not a mutable system prompt. These markers offered reuse opportunities on a supporting route; only provider usage could show that a cache write or read was granted.

Stable bytes do not establish a cache policy

After repairing the lens, the historical investigation varied request size. A locally stable 99% prefix produced no cache writes at 8,000, 10,000, or 18,000 tokens. The first request above roughly 22,000 tokens was cached, and the next read back 22,144 tokens. The investigator compared that with the then-documented 1,024-token minimum and described an apparent floor.

Seven requests did not establish a universal threshold. Request size changed alongside session history, and the lens could not see which provider-side entries existed. Padding future

prompts to reach 22,000 tokens would have turned an unexplained observation into a spending policy. Trimming can trade fewer input tokens against less reuse, but deciding which is cheaper requires actual usage and applicable prices.

Nor did size explain the whole record. The expensive morning's requests contained 55,000–73,000 tokens. A first-party client using the same OAuth token on a different endpoint recorded 61–85% hit rates, including 12,160 cached tokens on the first request of a fresh session. The record ended with a provider bug report and a route still too expensive for that workload. An expected fix within weeks was a hope, not a delivery commitment.

The live plan route also exposed assumptions shared by a parser and its fake: rejected explicit-cache controls, absent Content-Type, empty terminal output despite completed item events, and missing tool-call provenance. Each repair exposed the next mismatch. Chapter 16's separate second-edition run reproduced its own cache-options rejection and empty-terminal-output failure. Its live header was not retained; captured-body and controlled-header evidence supported the missing-header parser condition instead. Those records cannot be combined into a wire capture that never existed.

A fake and its client can agree because they share the same mistake. Captured provider behavior can correct the fake, after which a focused regression can catch the defect without another paid request. The meter supplies usage categories, the lens explains local changes, and provider counters establish granted reuse. None substitutes for the other two.

Reaching the screen, and reaching the supervisor

The historical agent had a GUI and a documented MCP server but no active path to inspect its own browser. A successful stdio client check had not established that connection. The repair supplied two directions: a loopback TCP relay let external MCP clients drive the GUI, while a local `view_gui` builtin requested a snapshot over the existing WebSocket. A recorded live check then had the agent correctly name the browser's buttons and tabs.

Before that worked, the tool had been registered and unit-tested without ever reaching a model request. The registry normalized `view_gui` to `viewgui`; the skill filter compared that key with the declared name `view_gui` and rejected it. Builtins survived accidentally because their constructor stored raw names while registration stored normalized ones.

Checking the declared name repaired filtering, then broke the MCP connection path, which recorded enabled tools using the normalized key. A cross-chapter check caught the regression, and the second repair aligned that path too. A normalized lookup key is not a public identity. Fixing one end of an inconsistent convention can reveal the other end's dependence on it.

The supervisor needed a different kind of visibility. Bill listens to narration through a screen reader at high speed; it lets him catch a mistaken approach before it becomes a sequence of edits. The changelog records a silent session followed by a hard reset that also discarded his uncommitted work. Missing narration made review harder. Destructive recovery added a separate failure. Interrupting the turn and inspecting the diff preserves more options, though the Agent and other jobs remain alive and still need supervision.

That implementation added a model-specific visible-text gate. If a round contained no visible text, the actor rejected every tool call before execution and returned a result for every call, preserving call-result pairing. Rejecting after execution would have been too late. The errors explained the rule and a concrete way forward, and asked the model to report surprises that might expose a mismatch.

This was a historical enforcement mechanism, not a new requirement here or a demand for private reasoning. Useful progress explains the intended change and its reason. As Chapter 16 establishes, public reasoning summaries and ordinary assistant explanation are separate streams; one cannot be assumed to supply the other.

Other observed mistakes produced explicit engineering standards. Deleting a comment that recall was hooked in two places removed the only explanation of an invariant while leaving both dependent call sites intact. Unformatted edits and stray backup files caused different harms. The resulting standards included checking output rather than trusting exit status, anchored edits, preserving explanations until their conditions are gone, and never weakening tests. Each rule carried a reason, so an exception had to answer an argument rather than merely ignore a slogan.

A file that actually changed

The first-edition account reports that Ensemble edited chapters describing itself, including a sub-agent voice pass. That is evidence of use, not a guarantee that each regenerated edition improves, or proof that the second-edition agent had completed the same crossover.

The second edition began with a smaller test: could Ensemble run Bill's selected model, repair a real program through tools, and leave a result another process could verify?

For the October 11, 2026 run, the selection was `gpt-6-astra` at high effort. Its existing table entry still chose Chat Completions. The documentation cited in the source required Responses for Astra function calls, so moving that row to the existing renderer was the entire runtime change. Effort settings, credentials, and stream handling already existed. The provider seam made the change small, but a model name alone had not selected a usable route.

The task was a CSV spending report that split lines on commas and summed amounts as binary floats. A merchant named `Books, Inc.` broke the field split; a quoted newline broke the line split; decimal currency and refunds exposed the arithmetic. Six tests fixed expectations before the agent began, including ordinary CSV quoting, escaped quotes, blank rows, exact merchant totals, and the existing sorted, tab-separated, two-decimal CLI output.

The CLI session read the source and tests, then ran them: three failures and three errors. Before editing, Astra explained the CSV and precision defects. It replaced manual parsing with the standard library's `csv.reader`, opened the file with `newline=""`, used `Decimal` arithmetic, and skipped blank rows while retaining the output format. The same six tests then passed. An independent rerun of preserved before-and-after directories reproduced both outcomes, and the test file was byte-identical. The source diff made the actual repair inspectable rather than leaving it inside the model's report.

Five requests completed in 27.46 seconds. Requests and terminal responses consistently named Astra/high. Subscription funding was selected explicitly, with no API key configured. The CLI exited normally after EOF and saved its conversation. The

endpoint was established through configuration and transport source, not an independent capture of the HTTPS URL or authorization headers.

The run produced 161 ordinary text deltas and no public reasoning-summary deltas despite requesting detailed summaries. Every response reported zero cache grants and zero reasoning tokens. Those fields do not measure private mental activity, and the successful repair does not establish cache savings. The ordinary explanation arrived before the edit, which supplied the supervisory visibility this task needed.

This was one bounded coding task, without a steering or GUI demonstration. It established a real edit, unchanged tests, independently reproducible results, and normal shutdown. It did not establish general project reliability or authorship of the next edition. The six cases narrowed what had to be trusted; the preserved diff let someone other than the agent judge the change.

Chapter 18: Giving the Agent the Web, and a Status Line

An agent that cannot reach the web can explain what it learned during training, but it cannot verify what a page says now. An agent that cannot inspect its own usage has a different blind spot: it can perform work without knowing what that work consumed. This chapter adds both kinds of visibility. The hard part is not giving the model more text. It is making the origin, authority, and limits of that text clear.

A fetched page is external evidence, not a new instruction. A status report is a snapshot of measured work, not the model's estimate of its own effort. Both features depend on paths already established in Ensemble: skills grant capabilities, tools return correlated results, jobs retain work that outlasts a wait, and owned objects provide access to the state they govern.

Web access belongs to the agent

Some model APIs offer native search. That is convenient until the agent changes providers, or uses a model whose API does not offer it. Search attached to a particular model is a capability the application cannot reliably carry across that change. A search tool dispatched by the agent has a different owner. The model chooses to call it, but the application controls its availability and execution.

MCP provides the bridge to an external search service. The agent initializes a connection, discovers the server's tools and their schemas, and converts those definitions into the format used for model requests. A call follows the same bridge back to the server, and its result returns to the conversation as the result of that call. The agent does not need a separate adapter for every search provider's API. It does need a trustworthy transport and an explicit policy about which discovered tools it will expose.

That separation mattered in Bill's recorded investigation. The open-source candidates included `ddgs`, a metasearch library with a built-in MCP server that required no credentials. Three manual tests looked promising; a larger set of fifteen succeeded only 73 percent of the time. The failures offered little explanation. Another candidate, `crawl4ai`, had no MCP entry point or module in the tested pip distribution. Its MCP support existed in the Docker deployment as an HTTP/SSE endpoint. The packaging mismatch mattered: installing the library did not provide the advertised server.

A clean probe of Firecrawl's hosted endpoint succeeded ten times out of ten, then the keyless endpoint failed often enough during sustained use that Bill moved to a keyed endpoint. The story is not a permanent ranking of services. It shows why discovery, installation, and sustained operation are different evidence. A short successful streak does not establish availability, and a failure that names its cause is more useful to an agent than an opaque refusal. Diagnostics determine whether the next attempt can change anything relevant.

The chapter's choice of a hosted service also changes the trust boundary. Queries and requested URLs leave the machine. The bridge sends selected tool arguments rather than automatically uploading the conversation, but the model can still put private conversation text into an argument. A smaller request surface is not a guarantee of harmless disclosure.

A connection is more than a POST

The earlier MCP bridge could communicate with a subprocess over standard input and output. A URL transport preserves the same client-facing lifecycle—start, send, receive, and close—but replaces pipes with HTTP. That small interface keeps the client independent of the transport. It does not make the two transports behave identically underneath.

HTTP success alone does not identify a useful tool reply. A server can return an ordinary JSON response, or a server-sent-events stream containing JSON-RPC messages. A single SSE event can span several data lines. A notification can receive an empty acknowledgment, often an HTTP 202, without producing any message for the client. Treating every response body as one JSON document breaks those legitimate cases. The transport must recognize framing before the MCP client can interpret the message.

`Send` parses the POST response and queues its messages; `Recv` drains that queue. One POST can enqueue no replies or several. A fixed-capacity queue can block `Send` if the consumer cannot drain it until the POST finishes. The implementation uses an unbounded queue with a condition variable to avoid that deadlock; the required behavior does not mandate that particular data structure. This POST-based path does not implement every optional transport capability, such as GET streams and resumability, or claim complete conformance to the current standard.

Initialization can also establish a session. If the server returns an `Mcp-Session-Id`, later requests must carry it. The request's Accept header needs to admit both JSON and event streams. These details explain a common half-success: initialization

appears to work, but the next operation fails because the transport discarded the identity or representation negotiated at the beginning.

Credentials belong on this transport side of the boundary. The skill's `auth-env` field names an environment variable whose value becomes a Bearer token in the Authorization header when the connection is made. The skill file carries the variable's name, not the secret. If the variable is unset, the transport logs the degradation and attempts an anonymous connection; the server decides whether to accept it. A silent downgrade could hide misconfiguration behind a low rate limit. The configured token must also be redacted wherever a peer reflects it, including error diagnostics and successful RPC content. Successful retrieval is not permission to copy a secret into history.

Cancellation exposes another difference from pipes. Stopping an HTTP request does not necessarily wake a goroutine blocked in a reply-queue wait. In the chapter's acceptance review, close initially appeared correct because the MCP client stopped waiting. It had a separate quit signal, while the underlying transport receiver could remain blocked. Testing the receiver directly exposed the unfinished shutdown. Close must end both the network operation and the receiver's wait.

This lifetime is separate from the agent's callback wait. A fetch that exceeds a short wait becomes a running job whose result can be collected later. It is not cancelled merely because the immediate caller stops waiting. Cancellation, timeout, unload, or session teardown can end the work; an ordinary running notice cannot stand in for its result.

Grant the capability, not the page's wishes

The web skill describes the search server and grants search and scrape operations. Loading it makes those tools available; unloading it removes them. Discovery is broader than authorization: the server may advertise many tools, but only the granted names may be dispatched.

Model-facing disclosure and dispatch enforcement serve different purposes. Hiding a definition discourages a call; refusing an ungranted dispatch prevents one. A model that guesses a tool name must not bypass the skill boundary. Conversely, a skill file sitting on disk is not necessarily loadable. The primary skill must allow it. Progressive disclosure narrows the tools available during unrelated work, but it is not human approval: a model allowed to load a skill can be persuaded to do so.

Catalogue presence is not itself a grant. Anthropic can receive deferred definitions in the request's tool catalogue; the paired `load_skill` result's inline `tool_reference` discloses a definition to the model. That reference is content, not a cache marker. In the October 22, 2026 live run, the provider rejected a request with HTTP 400 because a definition combined `defer_loading: true` with `cache_control`. The correction puts the tool-cache breakpoint on the last non-deferred definition, or omits it when none qualifies. It leaves the catalogue, its order and defer flags, paired references, and system/history cache markers intact. Changing cache eligibility must not change callable authorization.

The first recorded live probe found exactly this distinction between installation and availability. The default primary did not list `web-search` as loadable. A test primary did, so the isolated

tests passed while the installed agent could not use the feature. The mechanism was enforcing the supplied policy correctly; the delivered policy was incomplete.

Once the page arrives, it must remain a tool result tied to the initiating call. It does not become a user message or a system instruction. Keeping that envelope preserves provenance and gives the model a basis for treating the bytes as retrieved evidence. It cannot guarantee that the model will respect the distinction.

A page can contain text telling the model to ignore prior instructions, reveal context, or run a command. The dispatcher can refuse a tool that was never granted. It cannot prevent every misuse of a granted one. An attacker does not need a file-writing tool if a search query can carry private text to an external service. Source labels and warnings help the model interpret content; they do not turn the model into an enforceable security boundary.

Bill's reported practice is therefore sparse, supervised searching. Human observation provides a chance to intervene, not a guarantee that an attack will be recognized. Keeping credentials out of model context, limiting enabled tools, and minimizing disclosed query content each reduce a different exposure. None makes arbitrary retrieved text trustworthy.

What actually reached the model?

Search identifies candidates. Fetching lets the agent inspect a chosen source. Neither operation succeeds merely because some recognizable text appears somewhere in the conversation. The requested URL or search phrase may already contain that text. The important path is from the server's response, through the

correlated tool result, into the next model request. Multiple returned content blocks must survive that path rather than silently collapsing to the first one.

Error handling needs the same attention to origin. An HTTP failure connecting to the MCP service is different from an MCP tool reporting failure inside an HTTP 200 response. The former can be an authorization or quota refusal from a reachable server, not just a dead host. The latter carries the tool's `isError` flag and diagnostic, which must not be reformatted as a successful answer.

A third case lies beyond both: a successful scrape operation can return a failed web page. In the original probe, Firecrawl returned a page's HTTP 404 content with `isError: false`; the page status appeared in metadata. Checking only the tool-error flag would let the agent confidently summarize “Page Not Found” as documentation. Transport status, tool status, and fetched-page status describe different operations.

The second-edition live run exposed a further break after a real scrape had completed. The primary could load the web skill, but lacked `wait_for_job`. The model received a running-job notice while the finished page waited in the job store. The process exited normally without answering the question. More fetching machinery would not have helped. The missing capability was the existing means of collecting the result.

After that correction, one search, one scrape, and one wait produced an answer supported by the official Go download page. The fetched page was about 1.8 megabytes; the model received a selected 12,000-byte portion with truncation disclosed. The relevant filename and HTTP-200 metadata were in that portion. A fetched page, a saved job result, and the bytes shown to the model are three different artifacts. Saving the complete page does not give the model knowledge of omitted regions.

A later recorded CLI run, on October 22, 2026, completed the web task while switching between two same-provider models and checking usage. It used an Anthropic API key, not OAuth, and did not exercise the optional GUI. These are bounded observations about the recorded runs. Their release filenames are not current release advice, and their successful requests do not establish today's service availability.

Five numbers, and the missing path to them

The status tool asks for modest information: the selected model, counts from the last response, session totals, cache hit rate, and estimated dollar cost. The first-edition investigation found the arithmetic already implemented. The tool simply could not reach most of its inputs.

Bill had expected each child object to retain a path to its actual parent. That path should have made root-owned state reachable without separately passing each fact to each consumer. Instead, the historical binary constructed objects side by side, then supplied individual dependencies and closures whenever one needed another's state. A clean composition root existed, but nothing called it. Reading that unused constructor would have produced a reassuring account of an architecture the executable did not use.

The investigating agent initially concluded that the shared package lacked an Agent interface. Bill challenged the diagnosis. The interface existed under the name `Host`, but exposed only logging methods. Its name and narrow original purpose made it

difficult to recognize as the place to extend access to the owner. The cited declarations were real; the explanation built around them was wrong.

That misunderstanding had consequences. Each new feature added another side path instead of extending the ownership relationship. A model getter supplied by closure could serve one consumer, but it did not restore a general route through the owning objects. The status tool finally made this accumulation visible because it had no convenient object on which to attach another closure.

The old architectural check had not protected against it. It searched for the words `Logf` and `Host`, not whether a dispatched tool could reach its actual owner. The words remained while the property disappeared. The same distinction applies to a constructor: its presence proves little unless the delivered executable uses it.

Those are historical defects, not instructions to recreate the old tree. In the accepted implementation, the tool-call context reaches the owning Agent's Engine; that Engine exposes its Agent, and the Agent exposes Ensemble. The status handler obtains the Engine's snapshot; it does not create a second counter or a separate status service. Shared interfaces keep tool code from importing the concrete Engine and actor implementations. The headless and GUI entry points use the public construction path.

An interface field alone does not establish that relationship. A credential provider supplies a service; it is not automatically the parent of the configuration that uses it. The useful question is who owns the state and whether this particular call reaches that owner.

Counts retain their history

The historical cost display applied the currently selected model's price sheet to all session tokens. Switching to a cheaper model therefore made earlier work appear cheaper too. The arithmetic was correct for a single model; its inputs had lost the distinction needed for a mixed-model session.

Storing a dollar total would hide a different problem: prices can change while measured token counts remain history. The accepted meter keeps counts separated by vendor, request model, and price tier, then derives cost from those buckets. A change in the selector does not move old usage into the new model's bucket.

If any contributing bucket lacks a price, the full dollar estimate is unknown. A sum that silently excludes unpriced work looks more precise than the evidence permits. Returning to a priced model does not erase the unpriced work already recorded. Counts remain reportable even when cost is not. And a model-token estimate is neither a subscription invoice nor the bill for a separate search service.

Cache counts need equally explicit scope. Cache reads are measured provider usage, not a reward inferred from reusing a prompt. The hit rate divides cache-read tokens by total input: fresh input, cache writes, and cache reads. Cache writes also contribute to cost. Omitting them from the denominator inflates the apparent reuse rate.

The last-response and session views answer different questions. The last-response fields describe the latest completed main request; the accumulated view describes recorded work across the session. The status call itself occurs during inference. It sees the counts recorded when the tool executes, before the model has produced its final answer about those counts. A later

GUI meter includes more work. Comparing those two observations as though they were simultaneous creates a false discrepancy.

The recorded browser exercise illustrates this without needing a guessed dollar value. Two completed turns used different selected models, and the actual provider requests carried those names. Earlier usage survived the switch. The final visible meter then advanced beyond the second status result because the answer added tokens. Both tool results reported unknown cost because the model rows had no prices. That run established routing and measured counts, not priced accounting; separate controlled checks covered known tariffs.

Selection and lifetime remain owned operations

A real model switch is not the model saying it has changed identity. The CLI's `/model` command and the GUI's model field invoke the Agent's operation at a turn boundary. It rejects an empty name or a busy turn, changes the selection, and clears the old resolved identity. Conversation and usage survive, as do provider, endpoint, credentials, and billing configuration. This is a same-provider model change, not an arbitrary provider migration.

The effective selection must govern the next provider request as well as the displayed label. It is session state rather than a saved preference: restarting uses launch configuration. A mixed request to change both model and preferences is refused before either is applied, avoiding an ambiguous partial update.

Ownership also has a concurrency dimension. During the historical status investigation, disconnecting a GUI client could race with a notification sender. The sender took a snapshot of clients under a lock, released the lock, and then sent. Teardown could remove a client and close its channel between those steps. A nonblocking send did not protect it: a default branch handles a full channel, not a closed one. Sending to the closed channel still panicked.

The remedy was a coordinated lifetime, not another getter or a catch around the send. The accepted GUI keeps the data queue open, coordinates client iteration and removal under the server mutex, and uses a separate completion signal to stop and join the writer. A channel need not be closed to be garbage-collected. The historical tests became informative only when they arranged overlap between sending and teardown and could expose the restored defect; many repetitions without that overlap had passed falsely.

An owner chain answers where state comes from. A lifetime protocol answers whether it is still safe to use. Neither substitutes for the other. With both intact, web tools can return evidence without acquiring authority over the agent, and status can report measured work without inventing a second history of it.

Chapter 19: The Sandbox

In Bill's October 2025 account, the CodeRhapsody secrets manager did exactly what he designed it to do. Credentials lived in an encrypted store; an approval dialog controlled their release. When Claude Sonnet 4 needed the sudo password, it ran `cat` and asked permission. Bill was following the reasoning and clicked Approve before noticing the consequence: the password had become a tool result. The agent never needed to ask again. Three hours later, he found it committed to Git. Scrubbing the history took longer than writing the secrets manager.

The mistake was in the data path. Approval released the credential into a conversation from which it could be repeated, logged, or committed. A better dialog might have caught this particular decision, but it would still have depended on Bill noticing it. He also reports mistaking Codex's unrestricted mode for autonomy inside a sandbox while starting the second edition. Repeated approvals had made full access attractive; the name had not supplied the missing boundary.

Chapter 18 gave the agent access to untrusted web content. Combined with private data and external action, that forms Simon Willison's "lethal trifecta," described in June 2025. A poisoned page can ask the model to read a secret and send it elsewhere using legitimate tools. No exploit in the tools is necessary. This chapter restricts what those tools can do even when the model requests the wrong thing.

The runtime is trusted; its instructions are not

Ensemble's design does not put the entire binary in a process jail. The binary is trusted host code, selected and compiled by the operator. It holds credentials and attaches them to provider requests. The untrusted material is model output, including tool calls influenced by retrieved content. The boundary confines the operations that output can request while leaving the host able to communicate with the model.

That is a specific threat model: a poisoned conversation driving legitimate tools. Compromised host code and operating-system escapes are outside it. The host, its chosen root, and its trusted configuration must remain outside the model's control.

Keeping secrets out of context closes one disclosure path, but not all of them. A model can ask a command to copy bytes it never sees. Conversely, denying network access to commands does not prevent the trusted host from sending model requests or a granted web tool from sending a query. Files placed inside the workspace and capabilities granted to the agent still determine what can leave through those routes.

Two kinds of operation need enforcement. File tools run inside the trusted Go process, so they use confined file operations. `run_command` launches arbitrary programs, which do not call Ensemble's path checker. Those processes and their descendants need an operating-system sandbox. Neither mechanism can substitute for the other.

The policy has one owner. The existing public `Config` carries `SandboxRoot`, `SafeMode`, and `EnableWebSearch`; Sandbox retains its required Agent parent and obtains the configuration through it. Tools reach that Sandbox through the Agent. A copied

root in a helper or callback would create another policy value that could become stale. An empty root selects explicit full access; a nonempty root selects confinement.

A checked path is not a confined operation

For a file-tool request, a relative name is interpreted under the sandbox root. Canonicalization removes traversal components and resolves symlinks, including existing ancestors when the requested file has not yet been created. The canonical candidate must be inside the canonical root as a directory relationship, not merely share a string prefix. `workspace-other` is not a child of `workspace`.

Canonicalizing only the candidate caused an early failure described in the source. On macOS, `/tmp` resolves to `/private/tmp`, and `/var` to `/private/var`. A root created under an alias no longer matched its own resolved contents, so legitimate accesses were refused. Both sides of the comparison need the same interpretation. The root is canonicalized at use rather than trusted as a permanently resolved string.

Even a correct comparison cannot make a later ordinary open safe. A process can replace a symlink between the check and the open. Worse, an interface that only returns a checked name invites callers to check one string and accidentally pass the original string to `os.ReadFile`. The common interface therefore exposes operations such as `ReadFile`, `OpenFile`, and `ReadDir`, not just validation.

The source implementation uses Go 1.25's `os.Root` file, directory, and rename operations, passing relative names to an opened directory root. The traversal resistance belongs to the

operation itself, rather than to a prior promise about a pathname. Go's traversal-resistant API explanation, cited by the source, explains that distinction. This uses the standard library's protection; it is not a new guarantee against every hard-link, mount, or hostile-native-code scenario.

The same reasoning covers less obvious reads and writes. A command can replace `save.json` with a symlink before the trusted host saves state. Media reads and later reads of retained job output can cross the boundary just as ordinary file tools can. They need the same confined operations. Application-selected skill files are a separate trusted input and may live outside the workspace; their location does not grant file tools permission to read there, nor does confinement remove secrets mistakenly placed in that trusted corpus.

Refusals have a smaller information boundary of their own. They report the caller's supplied string, not a resolved host path or expanded working-directory diagnostic. An error should explain the refusal without teaching the model the filesystem layout it was denied access to.

The shell needs a kernel boundary

Redirecting `HOME` and removing environment credentials changes a command's defaults. It does not stop an absolute read of `/etc/passwd`. On macOS, the chapter uses Seatbelt through `sandbox-exec` to constrain the child process. Descendants inherit the sandbox, so adding another shell or launching Python does not bypass it. The trusted Ensemble process remains outside, and command output returns through the existing job supervisor.

The profile denies command network access. Applying a single network-enabled profile to the whole process tree would lose the distinction between the host's provider requests and a command's attempted connection. Confined commands on unsupported platforms fail closed with an unsupported-platform error; this chapter does not implement Linux or Windows confinement. `sandbox-exec` is marked deprecated in the macOS man page, so the source's successful local use is not evidence of indefinite availability or portability.

The implemented profile starts with default allowance, then denies file reads and writes globally and adds specific exceptions. An earlier deny-default experiment failed before `main`; importing `bsd.sb` did not rescue it. That showed missing startup allowances in that experiment, not the impossibility of a deny-default design.

Useful commands need more than their workspace. The profile allows reads of the root directory entry, system paths such as `/usr`, `/bin`, `/System`, `/Library`, and `/dev`, and the TLS trust store under `/private/etc/ssl`. It permits writes to selected device files, including `/dev/null`, which Git opens for both reading and writing. Required ancestor and system-selection metadata is allowed without opening sibling data. The workspace is readable and writable. This is not `chroot /` still names the host root, and the system exceptions are real exceptions to workspace-only access.

Failures in those allowances can look like unrelated defects. A missing TLS configuration read looked like a network problem. In the restart record, Python and Git could not inspect `/var/select/developer_dir`; narrow metadata allowances restored both on that host. That result did not establish that every added literal was necessary, and shell-selection warnings remained even after the commands worked. The correction did not justify opening the whole selection subtree.

The Xcode shim cache exposed the opposite danger: allowing too much to make tools work. Git, Python, and Clang can pass through shims that use randomly suffixed `xcrun_db*` files in the per-user temporary directory. Allowing that whole directory also allows temporary workspaces belonging to other processes. A narrower-looking rule still failed during the second-edition review: a temporary-directory subpath rule combined with a basename pattern allowed a planted `xcrun_db-private-marker` inside an unrelated nested directory. A real command read its contents.

The correction anchored the allowance to immediate cache files under the physical temporary directory, preserving Git and Python without admitting the nested marker. Location mattered as much as the name. Physical paths mattered too: Seatbelt matches `/private/var`, not the `/var` alias returned by the temporary-directory query. An earlier investigation changed both canonicalization and a trailing slash; isolating the changes showed that canonicalization fixed the failure and the slash was irrelevant.

Credentials stay with the transport

The trusted binary adds authentication at the HTTP boundary. Provider-specific headers carry the credential; the conversation carries the request and response content, not those headers. OAuth refresh likewise belongs to the credential provider inside the binary. The model neither receives the refreshed token nor participates in obtaining it.

A server can reflect a credential back in a successful response or an error. Logs are another route: a key written to `api.log` inside the workspace is now readable data. Known secrets must therefore be scrubbed before events, traces, logs, or model-visible

results retain them. The source's logger replaces the entire known value with `[redacted]`, while transport handling includes current and renewed Bearer-token echoes. Transport failures expose a generic safe error or the context's cancellation error rather than credential-bearing details.

Scrubbing after writing is too late; the model may already have read the file. Matching known strings is still only a backstop. It cannot recognize every encoding, split disclosure, or credential unknown to the logger, and it does not justify placing a credential store in the workspace.

Commands receive an allowlisted environment rather than a blacklist of suspicious variable names. A blacklist misses an unusually named credential and can retain executable startup hooks. The confined environment keeps selected locale and timezone settings, an explicit system PATH, and workspace-rooted HOME, TMPDIR, and Git configuration. SSH agent access and arbitrary inherited variables are absent. HOME redirection makes ordinary lookups local; the kernel must still refuse absolute outside paths.

Git illustrates why redirection is often better than another exception. Its real user configuration can contain tokens in URL rewrites, credential helpers, or includes that lead elsewhere. The source instead points `GIT_CONFIG_GLOBAL` at a minimal generated workspace configuration and `GIT_CONFIG_SYSTEM` at `/dev/null`. Local commits get an identity without credential helpers, rewrites, or includes. The recorded local `init`, `add`, `commit`, and `log` operations worked without exposing the user's configuration.

The original credential discussion proposed delivering a needed secret directly to a trusted process's stdin, with only a confirmation returned to the model. That would avoid the October 2025 failure's direct path into context, but the recipient could still echo it, so output would need protection too. This

chapter adds neither a `send_secret` tool nor a secret store. It establishes where credentials belong, not a new means of handing them to arbitrary programs.

Narrowing capabilities is an operation

Confinement, safe mode, and web access are independent choices. The application exposes them as `--sandbox`, `--safe-mode`, and `--no-web-search`, with corresponding configuration and environment inputs. Removing shell tools does not confine file tools. Selecting a root does not disable web search. CLI and GUI application defaults enable search, while an explicitly supplied `Config` is authoritative: its zero `EnableWebSearch` value disables it.

Safe mode removes `run_command`, `send_input`, `wait_for_job`, and `kill_job` from declarations and dispatch; the source's accepted implementation also removes `tool_limits`. Ordinary file tools remain usable. Hiding a declaration is insufficient if a model can guess the name and call it anyway.

The first edition's initial safe-mode test found exactly that kind of reassuring non-boundary. `RemoveTool` normalized `run_command` to `runcommand` and deleted a registry key that did not exist. The entry under its real name survived. Tools with underscores stayed callable, and normal successful-tool tests gave no indication that removal was broken. The relevant property was absence and refusal after removal, not whether the removal function returned without error.

Web search has a different gate because Chapter 18 supplies it as a skill, not built-in tools. When disabled, `web-search` is not loadable, its MCP server does not connect, and its tools are not

bridged in. This closes the named skill's route. It is not a general network policy for arbitrary custom tools or other MCP servers.

Later child agents need to inherit these choices without silently regaining authority. `Config.Child(dataDir)` starts from the parent configuration, changes the data directory, and clears parent-specific log and save paths. Copying restrictions avoids the ambiguity of constructing a fresh configuration whose false Boolean might mean either “explicitly disabled” or “nobody considered this field.” Untouched fields already have the parent's values.

`Clamp(parent)` then returns a configuration no wider than the parent's and reports any attempted widening. A child cannot turn off a parent's safe mode, turn on search the parent disabled, or select an empty or escaping root when the parent is confined. A canonically contained narrower root is allowed. The Boolean directions differ: safe mode is a restriction and narrows with OR; web search is a capability and narrows with AND.

Enforcement and diagnosis are separate. Even if the caller ignores the reported error, the returned configuration is narrowed. The caller must still use that returned value; the method cannot protect construction that bypasses it. This chapter prepares the policy operation, not a fleet of spawned agents, and it does not claim that a restricted child's text cannot influence its parent.

What the observations establish

The source's first profile experiment appeared to pass every escape test because it denied everything, including reads inside the workspace. Positive controls exposed the useless result. Evidence for confinement needs both a refused operation and useful work that still finishes.

The outside fixture must exist and be readable by the host; failure to read a nonexistent file proves nothing. A network target must be reachable outside the sandbox and remain alive during the confined attempt. A compound shell command can end successfully after an earlier denial, so the last exit code alone does not establish each operation's outcome. Refused writes also need an absent file effect. These distinctions make the evidence about the boundary rather than about a model's description of it.

The source records earlier live sessions that completed file work and a local Git commit while refusing outside paths, symlinks, descendant reads, and command network access. Those sessions preceded the nested-cache and working-directory refusal corrections. Direct regressions covered the corrections; the account does not claim another paid session for them.

The restart record, dated October 22, 2026, describes Sonnet 4.6 sessions through the API-key CLI. Paired results and artifacts showed file work, Python, and Git alongside traversal, symlink, and outside-write refusals. A Python-to-shell-to-`cat` chain failed to read outside. A confined network attempt added no request to a reachable local peer, while the host continued provider communication. A known synthetic token was filtered from returned output and absent from retained logs. A narrower safe-mode session retained file work, refused a parent read, and omitted command and job tools from declarations. The completed sessions exited normally at EOF. These are recorded observations, not a new runtime verification; they did not exercise the deferred OAuth/browser flow or establish universal isolation.

Full access remains an explicit option for work that confinement would prevent. Bill reports using it during a late-2025 smart-device compromise at home. Unexpected speech from a Wiim speaker led him to change the wifi password and investigate. A supervised agent read configurations, scanned the

network, checked logs, and helped respond across a dozen devices. He reports a much faster response, without a retained timing measurement. Private data, potentially hostile logs, and external reconfiguration were all essential to the task.

That account explains the choice to accept broader exposure, not a proof that observation makes it safe. The sudo-password incident came from a person watching closely too. Where a confined workspace fits the job, the boundary takes a decision away from the poisoned conversation that neither an approval dialog nor a prompt can reliably enforce.

Chapter 20: The Ensemble

The first-edition executable reached its first spawn request and answered “no agent factory configured.” The fleet tests had supplied factories of their own; application startup had supplied none. The library could manage a child that the product could not create. This was not a missing convenience around an otherwise finished feature. Until the executable connected construction to the tool that requested it, spawning did not work for its user.

The GUI encountered the same boundary from the other side. After the first edition's architecture repair, Bill wanted the Artifact scroll for Homebrew-VTT. The component was reusable, but its Go package remained behind `internal/`, where an outside application could not import it. The repair had missed a boundary nobody had tried to cross. A terminal fleet and a separately shipped GUI look like different projects, but both depend on someone outside the component being able to reach a complete, supported capability.

Chapter 19 established confinement for model-requested operations, with policy in the existing Config and child permissions allowed only to narrow. Here that ownership extends to a terminal parent constructing children, alongside an outside application using the public GUI. The browser still presents one Agent. A fleet panel comes later; neither child construction nor its completion rules should depend on that panel existing.

A public boundary needs an outside caller

In the chapter's design, the root module owns Agents, Config, providers, conversations, tools, the stream actor, persistence and fleet management. The public `ensemble/gui` module owns browser-facing connections: the WebSocket bridge, settings handler, MCP relay and static assets. Its constructor, `gui.New(root, agent, log)`, accepts public values and returns a server with handlers and an explicit lifetime. An optional executable assembles them. Importing the core must not import the browser transport.

Public spelling alone is insufficient. A constructor is unusable to an outside application if its arguments have internal types, or if initialization still requires a private helper. The source's outside consumer constructs real public inputs, records a public event for replay, sends a prompt and reaches the relay. Those are distinct uses of the boundary. Successful construction cannot establish that replay works, and a rendered replay cannot establish that a new prompt reaches a provider.

The dependency graph needs the same skepticism. The first-edition GUI move left an indirect path from headless core tools through the MCP package to Gorilla WebSocket. Browser code had moved, but a WebSocket client remained in the headless protocol module. Separating that client removed the browser implementation from the core import closure. Checking a headless consumer alongside a GUI consumer matters: the latter is the positive control showing that the dependency check can actually find the transport. A clean import closure establishes dependency reachability, not runtime behavior.

The GUI constructor also needs a complete owner relationship. The first-edition Hub took a bundle of controls and acquired a current-model closure afterward. Reading the model live was necessary because an operator could switch it, but another caller could forget the late assignment. Moving the closure into a keyed callback struct did not make it mandatory: omitted Go fields still receive zero values. The accepted constructor instead checks the supplied root, Agent and logger, including that the Agent belongs to that root. Through the Agent it reaches configuration, Engine, history and pause state; Ensemble supplies observations. The server owns browser connections and frontend settings, not the Agent it displays. The application controls both lifetimes.

The public MCP relay accepts a listener chosen by the embedding application and forwards JSONL requests through a connected browser. Two callers may use the same JSON-RPC ID while replies arrive in reverse order; each must still receive its own response with its original ID. With no browser, the error is immediate and correlated. A connected but silent browser instead leaves work pending until a relevant connection or the server closes. Continuing to read the caller's socket detects disconnects and releases that caller's wait. Server closure closes listeners and joins connection workers. A no-browser error cannot establish these pending-work lifetimes.

Deployment includes assets and their owner

The chapter's deployment model embeds the shell, styles and JavaScript modules in the GUI package. A built executable can therefore run away from the checkout, while an explicit `--gui-`

`dir` selection lets a developer serve disk assets. The choice remains explicit: silently discovering a nearby source directory would make the same binary behave differently depending on where it starts.

Serving a path is not the same as serving the right asset. In the source's student run, an initial check accepted HTTP `200` from a directory listing that contained no GUI. Requiring the actual application shell exposed the defect. The response body was the missing evidence. Likewise, a settings acknowledgment shows that the control was received; the next provider request's model field shows whether a turn-boundary model switch took effect. Neither replies counted nor successful HTTP status codes substitute for observing the behavior being claimed.

`gui.Assets()` exposes a read-only filesystem because some callers need less than a server. The embed anchor stays private: Go requires that anchor at package scope, but exporting the variable would let callers replace its binding even though the embedded files are read-only. Returning `fs.FS` exposes the resource without that mutable public binding. A host can serve the Artifact scroll and renderer modules, mount the component in its own page and omit Ensemble's application shell. The shell owns its WebSocket connection, sends incoming events to the scroll and sends outgoing controls. The scroll and renderers own presentation. The extractions were substantially in place already; deployment should preserve that separation rather than invent a second UI protocol. A component-only page rendering a message while the application index is absent establishes a different kind of reuse from a full GUI session.

An embedding application owns more than HTTP registration. It supplies provider configuration, authentication, storage, logging, listeners and orderly shutdown. The source's small external application starts a fresh conversation; it is not evidence of resuming a persistent task. It prints newly allocated

loopback addresses to avoid accidentally observing another running instance. Such details make the observed boundary attributable to the application under examination.

That application must also distinguish the provider configuration resolved by a running Agent from settings saved for the next launch. Children inherit their parent's resolved, running configuration; they do not re-read environment variables or saved next-launch settings. Permitted model and local-preference overrides remain explicit. When Chapter 21 introduces saved provider edits, those edits must not silently switch a new child's billing or credentials while its parent retains the old values. Credentials remain application and Agent configuration, not routing fields in a model-visible specification.

The source records a second-edition deployment from a fresh directory: Chrome loaded the embedded shell and six supporting assets, and a provider-backed task wrote and read back a 42-byte file before normal exit. That is local evidence for the embedded application and that task. It does not establish component-only rendering or relay teardown, which have separate boundaries. Nor is the record a fresh observation of today's executable.

Children have separate state, not separate worlds

A terminal fleet begins with real construction. Each child is a full Agent with its own actor, tool visibility, conversation, save file and data directory. Sharing a parent's conversation and wrapping several calls in child names would not provide those properties. A failed child can be shut down and respawned without replacing its parent's conversation. The Agents share a process, however; separate state is not process-crash isolation.

Each Agent's manager tracks children, while an application-supplied `AgentFactory` constructs them. The factory receives the actual parent and child specification and returns an Agent. The manager loads its save, registers it and starts the task. In the restarted Edition 3 design, the existing public Config supplies the configuration role; a second configuration system is unnecessary. Application composition stays in the factory, while registration, parentage and lifecycle stay in the library.

The terminal installs the factory on Ensemble before constructing its Agents. The manager derives child IDs and paths from the parent; `NewChildAgent` checks the actual owner relationship and clamps the specification. Descendants reach the shared root through their real parent relationships, rather than relying on copied factory callbacks. A separate human sink makes the root's upward questions deliverable. An embedder that installs no factory retains its ordinary tool surface.

State lives under distinct child data directories, such as `cr/children/<id>/`. Each save contains that child's conversation, event log and tool state. Shutdown preserves the directory, and restarting the same child ID loads its saved conversation before starting the new task. That does not require a private task workspace. Several children may work on the same repository while their conversation and runtime files remain separate. File ownership and shared-edit coordination still need to be assigned. Independent Agent state is not filesystem isolation, and neither establishes freedom from conflicting changes.

Chapter 19's narrow-only rules apply at construction: a sandboxed parent cannot create an unsandboxed child, and a parent without web access cannot grant it. Runtime role instructions must agree with that construction. A worker's instructions should be composed from the authored base, not from an already-rendered root prompt that prohibits submission.

A worker choosing another model also needs that model's policy without duplicated or stale policy text. Successful construction alone cannot reveal an instruction that tells the child not to perform its required completion action.

A parent must remain able to answer

The fleet is a tree, not a flat collection of workers attached to the root. A child's routing ID accompanies, rather than replaces, its actual Agent parent relationship. A grandchild's question therefore goes to the child that delegated its work; it reaches the human only through the necessary parent steps. Hardwiring every parent to the root would make a shallow demonstration look correct while breaking recursive responsibility.

The human has a configurable address but is never a managed child. Uniform addressing does not grant authority to spawn, shut down or inspect the status of a person. Keeping the human out of the children map avoids making every lifecycle operation compensate for that false equivalence.

Questions suspend their askers. `send_message_to_parent` creates a pending question and waits for a reply. When the parent is the human, the root sink displays the question and retains its reply channel until terminal input arrives. Incoming messages retain sender identity. The question, pending state and reply routing must survive the interval when no model turn is actively producing text.

The first edition initially supplied the parent link only at submit time. A child could deliver its finished work, but it had to claim completion to ask for help. Bill caught that conflation: asking needs its own verb, available before finishing. A separate collision appeared in the student implementation when a parent and child with the same local name generated the same first

question ID. The child question overwrote a pending question to the human. The fix allocates IDs at the recipient's manager under the lock protecting pending questions—the scope in which replies must be unambiguous.

`respond_to_parent_message` can omit the opaque message ID when exactly one matching question is pending. The source describes this as removing a reliability tax: the model need not copy an identifier out of prose when there is only one possible answer target. More than one pending question still requires an unambiguous selection. Convenience cannot replace question identity.

Fan-in must accommodate this conversation. `join_agents` waits for the selected children to submit, or all managed children when the selection is empty. If a child asks a question during the wait, join returns a tool result so the parent can answer and resume joining. Otherwise the parent would wait for a result from a child that was waiting for the parent.

Human steering needs a different interruption. Input can wake the parent's wait while the underlying join job remains alive, allowing the same handle to be collected later. The terminal therefore must read input independently of the goroutine waiting for tool completion. The first-edition command loop did not: it promised steering in a comment while blocking the very reader that could deliver it. No amount of wake logic inside the Agent could compensate for that executable wiring.

Message wake is independent of the callback timer. Existing `tool_limits` supplies a one-shot override for the next call, not a per-tool-name timeout map. An interrupt with no current waiter is a no-op; leaving it pending would let a later wait consume a stale signal. Terminal `/reply` input answers a pending human question rather than becoming a new task, and the actor remains the single place ordering conversation changes.

Status and shutdown complete this ownership story. `agents_status` exposes the managed children and their current states. `shutdown_agent` recursively closes a child and its descendants and removes it from the Manager. It is a lifecycle operation, not a way to pretend that unfinished work produced a result.

A result contract is not a verdict

Delegated completion occurs through `submit`, not through a sentence saying the work is finished. By default its object contains a required string `result`, preserving the familiar text-child contract. A caller can instead provide a `submit_schema` through `spawn_sub_agent` for a particular child. The source's reviewer returns a summary and an array of findings, each with a path and boolean `fixed`. That gives the parent data it can consume without extracting structure from narrative prose.

This schema belongs to the spawned instance. It is validated and defensively copied before construction effects, and the factory's returned child must retain it. The same compiled contract supplies the model-visible declaration and validates actual arguments before persistence or completion notification. Ordinary grandchildren retain their default contract unless their own spawn specifies another. A parent's reviewer schema is not an inherited conversation-wide rule.

The chapter deliberately limits this validator to a declared subset of JSON Schema Draft 2020-12. The root must declare object type; nested schemas may be objects or booleans. Object and array constraints, exact numeric bounds, string bounds and patterns, enums and constants are enforced rather than merely advertised. Annotations insert no defaults and coerce no values. Unsupported keywords and all references are refused at launch;

schema resolution never fetches a URL or reads a file. Numbers must survive without rounding through `float64`. The schema itself needs validation, not just a check of its keyword names.

Admission limits bound schema and payload bytes, value-node counts, depth and numeric-literal complexity. They refuse rather than truncate. The combined canonical schema-by-payload byte budget can reject a pair whose members each fit their individual limits; it is a conservative work bound, not a CPU deadline. A real validator still has to enforce the admitted semantics. The public string convenience method enters that same validator and cannot bypass a custom contract. Arrays, booleans, nulls and nested objects remain JSON data through joins, status and durable storage; a string containing JSON is not an equivalent result.

Persistence distinguishes compatibility from an explicit contract. The default text path accepts and ignores extra keys, storing the normalized result object. An explicit custom schema preserves the complete object even if its shape resembles that default. Its JSONL row in `submit.md` includes version 1, the schema and the payload. An unknown version or malformed row is an error, not permission to silently return an older valid submission.

An invalid submission leaves the child unsubmitted and does not satisfy a join. The source's controlled reviewer example omits `fixed`, receives a validation error, then supplies boolean `false`. Inspecting the accepted nested payload and saved row establishes more than finding a marker somewhere in printed output. Retaining a default text child and an ordinary grandchild also checks that the custom contract has not leaked into unrelated instances. Defensive copies protect installed schemas and submitted payloads from later caller mutation. These are ownership properties, not just parsing details.

Shape still says nothing about truth. A well-formed verdict can be unsupported, and `fixed: true` can describe a change that was never made. The parent must examine the evidence and decide whether the child's claim is useful. Schema validation makes the handoff mechanically usable; it does not transfer the parent's responsibility for accepting the work.

The same precision applies to completion. A child with unfinished descendants cannot submit until it joins or shuts them down. If a delegated child becomes idle after a successful turn without submitting, the framework supplies a corrective wake. After exactly three consecutive wakes it stops trying and reports failure to the parent, with no fabricated submission. Submission resets the count; an unresolved failed child also prevents its parent from submitting until it is shut down. The source records an important correction: initially the exhausted child remained labelled running, so the parent could keep waiting for work the framework had stopped pursuing. Failed is a necessary state, not an inconvenient presentation detail.

A failed provider turn follows a different path. An HTTP rejection or execution error is reported directly, not fed into the three-wake loop. A refusal delivered as a successful model response may still be an ordinary unsubmitted turn; classification follows the runtime outcome rather than an interpretation of the prose. Wake attempts and escalation are recorded so that the missing submission is inspectable. The bound limits correction attempts, not the time to a good answer or the model's willingness to cooperate.

This policy is explicitly narrower than “all Agents submit.” The human-facing main Agent finishes with a normal chat answer, and a named peer addressed directly by the human is not automatically a delegated worker. Presence of a factory does not confer delegated status: correction checks the actual Agent parent

in the library's completion path. Explicit root submission remains available under the same admission rules, but ordinary chat completion needs no corrective wake.

Finally, accepted submission precedes quiescence. Publishing a result can wake join while the child's submit tool still needs to append its result and the actor still has completion and save work to finish. A caller disposing of that worker must wait for reliable completion and drain. Interrupting it inside the submission callback risks losing the final work that makes the accepted result durable.

What the executable and the observations can establish

The source's five first-edition wiring failures fit together. The application omitted the factory, omitted the human sink, failed to propagate the factory to children, required unnecessary message-ID copying and blocked input on the tool-wait goroutine. Fixtures had supplied enough surrounding machinery for the library paths to work. The delivered executable had not. A three-generation exchange reveals the propagation failure that a parent-and-child example need never encounter. A question to the human exercises a sink that child-to-parent messaging alone cannot establish. Library and executable evidence answer different questions.

A historical cross-chapter sweep found a separate defect: a package-level atomic message counter generated usable IDs but put per-fleet mutable state outside the fleet. Moving it onto the messaging state, under its existing lock, restored the owner. Successful routing scenarios had not exposed that scope error. Behavioral success is not evidence that every state lifetime is sound.

The source also records failures in the apparatus used to observe behavior. A cross-chapter comparison appeared to blame an unrelated one-line change for event-log failures. Absolute worktree paths and relative live-tree paths produced different results because the harness changed its working directory and resolved the log path differently. In another probe, the final report appeared but EOF never arrived: killing a `go run` wrapper had left a child holding the output pipe. Executing the built binary directly fixed supervision. Both cases required examining how the observation was produced before assigning a product defect.

The recorded terminal fleet gives a particularly useful split between delivery and compliance. Three children implemented and tested arithmetic functions. One asked whether negative integers mattered; the parent answered yes. A malformed shell command failed, and the model corrected it. That was task-level recovery, not the framework automatically retrying a failed turn. A terminal hint then woke the parent's join wait without killing the job. A later wait on the same handle collected all three submissions, and the parent ran combined assertions; independent execution of the files also passed.

Yet the requested `STEER_COBALT` marker was absent from the parent's final result. The exact hint appeared in a later provider request. Delivery worked; compliance did not. This historical run also used the earlier root-submit policy. Its parent's submission is not a demonstration of the current rule that the main Agent answers in chat while delegated children submit. Preserving those qualifications makes the record useful: it shows which mechanisms were observed without silently promoting them into evidence for changed behavior.

The successor inherits two reachable boundaries to maintain. An outside host must be able to use the GUI without private imports or accidental transport dependencies, and the terminal

must actually install the factory, routing, input and completion machinery that its fleet tools promise. Beneath both are owners with distinct lifetimes: an application closes its server, a Manager owns its children, an Agent owns its state, and a parent evaluates the work handed back. The next chapter can expose that fleet through the existing frontend without making visibility a substitute for those responsibilities.

Chapter 21: Managing the Swarm

A parent waiting for twelve children and a child executing a command can both report “running.” That label is enough to keep either from being mistaken for finished work. It is not enough to decide where an operator should look. Chapter 20 made delegation reachable from the executable; the browser still showed one conversation. Supervising the fleet meant asking its parent for reports and reconstructing the tree from tool results.

The management console makes that tree visible and opens the actual conversations behind it. Its harder job is keeping three facts separate: activity describes what an Agent is doing now, lifecycle describes the state of its work, and persistent identity determines what the application restores. An idle Agent may have submitted successfully, stopped without submitting, or finished an ordinary human conversation. A saved named peer may return after restart without any of its former task children. The screen has to tell those stories without inventing a simpler one.

Activity is an observation, not a verdict

The four activity phases are idle, thinking, tool call and waiting. Thinking begins with a model request. Tool dispatch reports the tool name and a short argument detail. A blocking fleet operation reports waiting, and the end of a turn reports idle. These facts come from the actor and dispatch path through the existing Observer, not from browser guesses about printed text.

Activity accompanies lifecycle rather than replacing it. A submitted Agent can be idle while retaining its submitted state. A waiting parent can have descendants changing files or running commands. Its phase is neither a security assessment nor evidence that earlier effects have stopped. The phase helps locate a question; the conversation and tool results help answer it.

The short detail makes a status useful without turning it into another log. The accepted implementation keeps the tool name separately and limits argument detail to fifty Unicode characters. A file name often distinguishes the call well enough; the full arguments remain in the conversation. Detailed status also includes measured token usage. Heavy input consumption with little useful output warrants inspection, but the counts alone cannot establish that an Agent is looping.

Persistent peers and task children

Ensemble owns the main Agent and a persistent registry of named peers. Each Agent's fleet manager owns its spawned children. Those are different relationships: named peers such as a researcher or reviewer survive application restart; task children are ephemeral and are not automatically recreated. Both can appear in the same browser tree without acquiring the same lifetime.

The registry, `ensemble.json`, records each peer's name, model, defining skill and creation time. It identifies the Agent; the Agent's own save holds its conversation. In the supplied command, named peers keep their saves and event logs under `agents/NAME/`, while the root uses `cr/` for Agent data and a separate default or explicitly selected save path. Task files still occupy the application's workspace. Separate histories do not provide private copies of the repository.

Creation builds and saves the peer, then persists its registry identity before announcing success. Empty model and skill selections inherit application defaults. Deletion refuses unfinished children and closes child admission before shutting down the peer and removing its identity and data. Without closing admission, a child could appear between the check and deletion. Application closure instead saves and stops peers while retaining their registry entries for the next launch.

Named peers have no Agent parent, just as the main Agent has none. Both normally finish in chat. Delegated children owe a submission, must resolve their descendants first, and receive Chapter 20's bounded correction when a successful turn stops without submitting. Displaying everyone beneath an Ensemble node does not make everyone a delegated worker.

The public startup path extends the owner already established in Chapter 20. CLI and optional GUI call `ensemble.New` and then `root.Start`, which installs recursive construction and the human route before restoring Agents. Existing non-fleet callers can retain `NewAgent`. Flags, authentication, terminal input and browser serving remain application concerns; conversations, tool visibility and jobs remain Agent concerns. One startup path reduces missing connections, but only exercising the executable can reveal whether it actually calls that path. A registry row proves that an identity was saved; the restored conversation proves more of what the operator meant by saving an Agent.

Settings have two application times

A persistent application also needs to distinguish the configuration it is using from the configuration it will use next time. The GUI command's versioned `ApplicationConfig` holds provider, model, billing, reasoning and endpoint settings

alongside external credential references, paths, runtime options and GUI preferences. It does not replace the public Agent constructors or impose a new loader on every CLI consumer.

Explicit `--config` mode is selected before environment builders run. Otherwise an inherited provider, skill or authentication variable could contaminate the file's meaning, or even prevent startup despite the file being valid. Relative paths resolve against the file's directory, not the launch directory. File mode accepts only the explicit port, GUI-directory and GUI-debug launch overrides, reports them, and rejects competing provider or Agent flags. Without a configuration file, legacy environment and flag behavior remains available.

Loading requires a supported explicit schema version and rejects unknown fields, invalid values, malformed or trailing JSON and embedded credential contents. Secrets stay in the referenced external key file or authentication store. They do not belong in browser settings payloads, flag defaults or help and error output. Saving validates first, uses private directories and a private same-directory temporary file, and replaces the file atomically. The saved in-memory snapshot is published only after success; a failed write or rename must leave the prior file and snapshot usable.

The settings view shows the file path and separates active provider values from saved next-launch values. Provider, model, billing and reasoning changes require an explicit Save and restart. They do not rebuild a live Engine, and newly spawned children continue inheriting their parent's resolved running configuration. Otherwise a saved edit could quietly give a child different credentials or billing while its parent retained the old configuration. Manual file edits also wait for restart; there is no implied live watcher.

Speech, theme, font and budget preferences remain immediate. In file mode, root preferences share the application file without overwriting provider fields. The old root `cr/settings.json` is ignored, not deleted; children retain local preferences. This mixture makes Save correlation consequential. Another tab's acknowledgment or an unrelated preference broadcast cannot mark a provider edit saved. Edits made during an outstanding save must survive its reply, and normalized credential paths must be handled without revealing their contents.

These are startup and interaction properties, not merely parsing properties. Supplying a constructor with correct values cannot establish isolation from a poisoned environment, correct path resolution from another directory, or the browser's distinction between pending and saved edits. Likewise, a successful Save acknowledgment cannot establish that the running provider changed; in this design it should not have changed.

Supervision stays inside the owner's fleet

A model-callable management operation resolves its target through the calling Agent's fleet manager, which contains its direct children. Foreign and nonexistent targets both return “unknown agent.” The caller does not receive a global map and then filter it. The trusted browser can walk the Ensemble to select conversations, but that broader visibility is not a fallback authority for model tools.

The same rule governs durable results. A submission file is readable through this interface only after resolving an owned child. Keeping a file after shutdown does not turn `get_submission` into a global archive lookup. This containment protects management operations; it is not process isolation or protection against every route through shared files and permitted tools. A rejection is meaningful alongside a successful operation on an owned child. A manager that refuses everything is contained but useless.

`wait_for_agent_change` observes a selected set of children for state changes, exits, parent messages, matching output or lack of progress. The accepted implementation snapshots those children, waits on the existing change notification or a short timer, then compares their state. The timer notices a busy child that stops progressing even when no notification arrives. This avoids adding a durable event-distribution system for a waiter that needs current state. Its work grows with the selected set; small fleet exercises establish no thousand-Agent performance claim.

An early version confused a notification with evidence about a particular child. A question from an unselected child advanced a manager-wide message counter, and the wait reported its selected child as the sender. Both requests were otherwise parked. Tracking message progress per sender fixed the attribution: the selected child's question still woke the wait, while the other child's question no longer impersonated its progress.

The wait has three distinct clocks. `timeout_seconds` bounds the observation job; `stuck_seconds` measures a busy child's lack of progress. The ordinary tool callback delay merely returns control with a running handle. It does not cancel the observation. A notification tells the waiter to inspect state, not what it is entitled to report.

Messaging separates steering from asking another question. `send_message` gives a busy child a hint between tool calls or starts a new turn when it is idle. Its recent output and cursor let the caller continue reading without replaying everything. `wait_for_agent` returns new finalized assistant text and tool output beyond that child's cursor, or returns when the child is idle. Streaming fragments belong to the GUI path: returning a fragment here and its completed replacement later would count the same output twice. `ask_agent` instead queues a distinct question and waits for that turn's result. Seeing new output is not equivalent to receiving the answer to that question.

Stopping also has more than one meaning. Graceful `shutdown_agent` waits for current and queued turns. `kill_agent` cancels the child's lifetime, including in-flight work, jobs and descendants, then removes it from the fleet. Application teardown needs the cancelling path so an unresponsive child cannot keep the process open. `interrupt_agent` preserves the Agent and independent jobs while ending its current turn: `now` cancels immediately, and `after_tool` stops at the next tool-report boundary. An acknowledgment proves delivery; observing the boundary and a later answer from the same Agent establishes the more useful behavior.

A durable result is not current completion

Chapter 20's default text submission and custom JSON contract remain intact across supervision. `Fleet.SubmissionData(id)` returns a typed `Submission`, with default text in `Result` and custom JSON in `Payload`. The

older string-only `Submission(id)` refuses custom data rather than silently forcing every caller to parse it. The tool returns `agent_id` with either `result` or a nested `payload`; status and joins preserve that object too. Where `ask_agent` must return text for a custom-only answer, it returns compact JSON rather than a blank successful string. Returned raw JSON belongs to the caller, not to mutable internal storage.

Accepted submissions append to the child's `submit.md`. Legacy rows contain the text result; versioned custom rows retain both schema and payload, so an older result keeps the contract that accepted it even when a later task uses another. Stored custom data is validated under that contract. Malformed rows and unknown versions are errors, not reasons to quietly return an older success. JSON string encoding keeps newlines inside a result distinct from the row boundaries.

Retrieval tries memory before durable storage, but it still starts with owned-child lookup. An explicitly respawned child can therefore expose a previous durable result while its new task remains running and unsubmitted. Restarting the application does not recreate that child automatically. The result may also survive as a retrieved tool result in its restored owner's history, which requires no live child at all to display.

Submission and disposal have their own ordering. Acceptance can wake a join before the submit tool has appended its result and the actor has finished completion and saving. Disposal must wait for reliable completion and drain, not merely the submitted flag. Invalid data remains unsubmitted; a durable old result cannot satisfy the new task's obligation. These rules keep result availability, task completion and safe teardown from collapsing into one misleading bit.

Selecting a conversation without disturbing it

The browser tree follows actual ownership. A synthetic Ensemble node contains the main Agent and named peers; each Agent contributes its children recursively. A local name alone cannot identify a node: `reviewer/helper` and `coder/helper` are distinct. The accepted tree carries an ownership-path key alongside the local ID, parent, persistence, lifecycle and activity. Initial complete-tree delivery and subsequent updates let a newly connected browser see current state without reconstructing all earlier transitions.

Selection belongs to a browser connection. Each Agent has its own retained conversation view, partials and replay cursors. Selecting it resets that connection's Artifact scroll, replays retained history and then continues live output. Another tab can keep reading another Agent while both receive tree changes. A global partial map would let response-local IDs from different conversations collide.

Changing the view also crosses a control boundary. Before selection, the browser cancels outgoing speech, clears its speech and typing pause causes, and sends `unpause` before `select_agent`. Otherwise leaving a conversation could strand its pending tool behind a pause gate. Late speech callbacks and queued output from the outgoing view must not alter the incoming Agent's pause. Correct isolation means the outgoing tool proceeds without another prompt while an incoming Agent that was already paused stays paused.

Structured submissions use the ordinary conversation surface, with tool arguments and results rendered safely as text while nested data remains intact underneath. Reselection and reconnect within the same application can replay the registered

child's result. After restart, the restored owner's history can show its retrieved result without resurrecting the ephemeral child. Finding a marker in a frame cannot establish this preservation; arrays, booleans and nulls must survive in the actual object.

Fleet observation must not make browser speed part of the actor's lifetime. Tree traversal can take several managers' locks, so a small coalescing worker performs it outside the synchronous Observer. Notifications request a refresh instead of making the actor wait while the GUI walks the hierarchy.

Selection exposed a more direct failure. Repeated clicks from a browser that stopped reading filled the replay queue. The receiver waited to enqueue history while holding the server mutex. Disconnect then needed that same mutex to finish shutdown. A slow tab had stopped the server.

The correction admits a whole replay as one queue batch and publishes the selected view only after admission succeeds. A connection unable to accept the batch is disconnected and can reconnect for a fresh replay. A larger buffer would merely postpone the blocked send; dropping the tail would silently lose conversation. Other connections and Agents must remain independent of that slow reader.

Visible creation and deletion complete the tree's controls. Creation supplies a name, model and defining skill to the existing Ensemble owner. Deletion displays the rejection when unfinished children remain. The GUI does not bypass the lifetime rule simply because a human clicked the button.

The task succeeded; the history was wrong

The source records a browser run in which a named peer, `console`, created one `helper`. The child implemented a small slug function, wrote three unit tests, ran them successfully and submitted. Its parent monitored activity, joined, retrieved the submission, independently reran the tests and wrote the outcome. The capture showed all four activity phases and selection between both conversations. An unused named peer was created and deleted. Normal shutdown and restart restored `console` and its conversation without another inference request; `helper` did not return.

That paid run used the earlier peer-submit policy. Its named peer submitted, whereas the current policy lets named peers and the main Agent finish in chat. The historical capture demonstrates neither that changed policy nor a fresh run of today's executable.

It did reveal something the passing tests missed. The screenshot showed idle activity beside `input_pending`, including after restart. Runtime work had ended, but retained history still claimed that tool results awaited another request. Ordinarily the next request consumes that pending input. Successful fleet submission ends directly, leaving no next request to conceal the stale state.

The correction records the existing `turn_ended` fact after the submitted tool batch drains and before completion and save. Replay can then present the truth already owned by history. Forcing every selected Agent to look idle would repair the screenshot by hiding genuine work. A separately parked request

supplied the necessary contrast: completed and restarted conversations should be idle, while an outstanding request remains `in_flight`.

The source's corrected behavior was checked through the real runtime, WebSocket selection and restart with a local provider peer. The earlier paid capture retains its bad label. There was no claimed paid rerun or migration of old incorrect saves. Successful task work, discovery of the display defect and the local correction check are separate pieces of evidence.

The original grader had an even earlier observation failure: CLI scenarios omitted the model, and its browser probe launched the headless command with GUI flags. Those launch failures established neither broken fleet behaviors nor working ones. The revised driver crossed the actual application boundaries, while focused controls addressed selected-child attribution, finalized cursors, interruption and slow clients. None makes a small coding task exhaustive evidence for the console. The useful distinction is concrete: supervision must preserve the state and ownership that make a report true, even when the work itself has already succeeded.

Chapter 22: Forgetting on Purpose

Bill initially mistook missing memory for a model-quality problem. In the spring 2026 account that opens the source chapter, CodeRhapsody interviewed a family assistant about improving its own supporting code. Both agents had persistent memory, and their suggestions built on earlier work. The same approach failed at his workplace. He blamed Gemini until he found a bug that had disabled memory there entirely. After the repair, the workplace agents improved tools and proposed procedural memory—the skills developed elsewhere in this book.

That experience establishes why continuity matters, not that any summary will provide it. A successor can read the code and still lack the reason a tempting approach was rejected. Keeping every conversation forever in the working request is expensive; deleting it can erase the very knowledge that makes the next session competent. This chapter replaces completed conversation with durable memory while retaining the original until its replacement is accepted.

There are three separate questions. Did the system replace exactly the intended source and recover it correctly after a crash? Did the replacement preserve useful meaning? Did the receiving model use that meaning correctly? A working file pipeline answers only the first.

A checkpoint selects work; it does not erase it

Chapter 14 removes tool traffic whose contents can usually be read or produced again. Conversation needs a different policy. A source file cannot tell a successor why an earlier implementation was rejected, what the user left undecided, or which failed experiment changed the plan. The retained event log may contain that history, but removing it from the working request still makes it unavailable to ordinary reasoning.

The source describes an older CodeRhapsody split: both `save_memory` and `handoff_task` cleared conversation, but only one fed the memory cascade. Bill recorded that gap on April 18, 2026; it remained open at the chapter's design session. The current design gives the operation one entrance, `micro_handoff`, rather than maintaining two clearing paths that must stay consistent.

The acting model writes a checkpoint note while it still has the task in view. After the successful tool batch, the handoff strips tool calls, results, opaque reasoning and private replay signatures. Visible dialogue and the original log remain. Memory then measures the remaining conversation and, if it exceeds its byte budget, selects the oldest whole completed segments and their notes. Unfinished work, skills and declarations stay out of the selection. Checkpoints at completed subtasks therefore define useful compression boundaries; checkpoints taken merely because a counter is high can leave half of two tasks in one memory.

The note is an opportunity to express reasons, not a way to extract private reasoning. A compressor cannot preserve a rationale nobody put into readable text. Useful notes explain

decisions, rejected alternatives, uncertainty and constraints rather than merely announcing completion.

One owned compression attempt may be running or awaiting commitment at a time. Further checkpoints can finish more work without launching overlapping summaries. The main Agent normally continues while the child works, with the selected source still available. Finishing or interrupting the main turn does not cancel the child. Explicitly killing that child or shutting down its owner does; a relevant policy change invalidates its uncommitted work. The near-window exception is a deliberate wait, not a return to synchronous compression everywhere.

Give the compressor a separate role

The source's October 14 trial explains why the compressor is a fresh child rather than a model call borrowing the main Agent's instructions. The earlier request inherited a rule telling the main Agent not to submit, then asked it to submit memory. The provider returned prose without a function call. The runtime refused the result and retained the source. Making that contradictory request asynchronous would have removed the delay without repairing the role conflict.

The current child inherits the owner's provider, model, route and safety ceilings, but not its coding primary, conversation or opaque replay state. It receives only the selected readable source, with role labels and handoff notes, delimited as data. Construction prevents skills and MCP servers from starting; the child has no recall, memory cascade, context forcing or public delegation. It uses the ordinary factory and ownership paths rather than becoming an unmanaged exception.

Its only tool is `submit`. The complete submission is an object with one string property, `memory`, and no additional properties —not a `result` wrapper or JSON embedded in a string. Public submission validation checks the shape; the owner also rejects blank memory. Both declaration and dispatch enforce the capability boundary. Asking a fully equipped model not to use file tools is not equivalent to withholding those tools, and guessed tool names must not bypass the restriction.

The child writes in the source owner's first person: retained experience should read as "I rejected this because…" rather than a report about a stranger. That voice does not promote the text into authority. Quoted instructions in the source remain data, as do the resulting memory bands. Putting a compressed account into system instructions would let text whose original provenance has been stripped acquire the Agent's authority. It would also rewrite the frozen prefix. Keeping memory in the conversation avoids both that authority escalation and unnecessary changes to earlier instruction bytes; it does not make memory immune to misleading content or cache costs.

An attempt has a finite correction budget: the current source permits at most three Engine sends and three tool batches. Existing bounded transport and authentication retries are separate. Prose-only completion fails; a mixed or multi-call batch is rejected as a whole, so a valid first submission cannot hide a forbidden second call. Provider length stops fail even when the partial JSON parses. An adequate output allowance matters: a desired compression ratio is no reason to cut off the explanation that makes a decision useful. The normal actor has no fallback to the deprecated direct compression call.

Bands organize memory without prescribing its quality

Memory appears between the frozen prefix and conversation, from stable to volatile: human-maintained `SOUL.md`, terminal `MEMORY.md`, the 64x and 8x tiers, then session memories. The tiers live in the owning Agent's corpus; another Agent has its own. SOUL never graduates. Session memory can move through 8x and 64x into MEMORY, where the cascade ends. The tier names are targets inherited from the design, not measured compression guarantees.

Each band has a distinct entry kind, so clearing dialogue tool parts cannot remove it accidentally. Band entries use `Seq: 0`; their canonical position is determined by band and file identity, not the event sequence that restored them. File identities use dates and same-day numbers independently of any log. Range filenames describe which memories a graduated file covers. This lets the corpus survive a particular log and makes restoring bands in different orders produce the same request.

Stable bands near the front have a better chance of retaining a reusable prefix. That placement is an opportunity, not a provider cache grant. Re-enabling an early band can still change a large suffix of the request.

Graduation uses a high and low watermark, measured in bytes. By default they are twice and once the band budget. Crossing the high watermark selects at least two oldest files, enough to bring the remainder down to the low watermark. The gap prevents repeated folding immediately after a successful fold. A single oversized file waits for another.

The source records an earlier mistake: requiring eight files meant a band holding five large memories could remain over budget indefinitely. Waiting for a nominal batch count required

the already-full band to grow further. The current selection follows byte pressure and whole-file coverage instead. Retirement must equal the selected oldest prefix, with nonempty progress; malformed, reversed or overlapping ranges cannot justify removing unselected work. The next cascade step begins only after the preceding replacement commits and its disk cleanup finishes.

A disabled destination blocks graduation with a warning and leaves source intact. Silently dropping the replacement or pretending the source had been reduced would make configuration changes destructive.

Accept the candidate where the owner can commit it

A valid submission is still only a candidate. Before accepting it, the actor waits for reliable child completion, drain and successful save. Seeing a submitted flag is too early: cleanup at that instant could interrupt the submission's tool-result tail. Finished workers leave the live Fleet, Ensemble and GUI views only after that boundary, while their traces and owner outcomes remain. A browser displaying the child returns to its owner; late observations must not recreate the dismissed child.

The owner then checks whether the child summarized the source that still exists. The selection captures entries, selected file bytes and coverage, the destination's previous bytes or absence, and the relevant policy generation. A human edit under the same filename invalidates the result. Turning a band off and back on also invalidates it, even though the final setting looks unchanged. New unrelated dialogue or an edit to an unselected later file does

not. Comparing against the entire growing conversation would reject useful work whenever the main Agent continued—the very concurrency this design permits.

Commitment occurs while idle or before a main render following a complete tool batch. It cannot revise a request already sent to the provider. Memory owns selection and corpus operations through its required Agent parent; the Agent actor owns attempt lifetime and commitment. History projects recorded facts, while the child's Engine calls the provider. These boundaries keep the background worker from editing live conversation or making History depend on filesystem guesses.

The commit sequence protects both sides of replacement. First, stage the candidate outside active memory scans. Then journal the task-correlated conversation cut or band retirement and its matching population. History applies the pair together: a cut without the replacement cannot remove source. Finally, install the destination, archive selected originals as `.md.retired`, record the terminal outcome and clear staging. Retired identities still reserve their file numbers after restart.

Writing straight into the active corpus would let startup load an unaccepted summary. Recording retirement without ensuring disk finalization would let startup reload originals the projection had already removed. A durable commit receipt bridges those states and survives snapshotting and log retention until disk work is complete.

Replay yesterday; read today's edits deliberately

Each `band_populated` event contains one memory's actual text and identity, not merely a filename. The first design proposed references, but neither the pure reducer nor the renderer owned a place to resolve them. A file could exist while the model received no memory. Recording bytes also distinguishes yesterday's delivered text from tomorrow's edited file.

A `band_depopulated` event records what leaves: a covered prefix for graduation, or the whole band when disabled. Managed cuts and populations share the attempt's identity; an ordinary disk rescan cannot accidentally complete a pending cut. Launch and outcome records describe attempts without turning replay into executable work. Reconstructing a request uses recorded cuts and bytes, never another model answer, current thresholds or a filesystem lookup.

After a process crash, an unmatched launch or uncommitted candidate is abandoned, with source retained. Replay does not respawn the worker; a later checkpoint may try again. A committed replacement with unfinished disk work is different. Recovery finishes installation and remaining archive renames before startup scans, re-enable, recall or the next main render. Otherwise the conversation could show the committed memory while retrieval searched an older corpus.

Conflicting human edits, missing candidates or unreadable files produce an explicit recovery error. Preserve the conflicting bytes and receipt rather than overwriting them to make startup appear successful. The hold remains interruptible and can clear

once repaired. This is process-crash ordering, not a promise of power-loss durability or a transaction against arbitrary concurrent editors.

Disk files remain the human-editable authority for current memory. Disabling a band removes context entries, not files. Re-enabling scans them and records each current text. Normal startup does the same after recovery, including for bands that remained enabled. Pure offline replay still reproduces the old recorded text without touching disk. Those operations answer different questions and need not disagree.

The source recounts an identity-only comparison that made corrected memory files ineffective: replay restored the old text, and the rescan treated the unchanged filename as sufficient. Comparing content as well as identity fixed it. Testing only disable-and-enable can miss that defect because disabling has already removed the old entries; an enabled-band restart exposes it.

The read-only `recall` tool searches the owning Agent's corpus, not another Agent's files or a transcript file as a shortcut. It can find retired originals and disabled tiers; SOUL participates only while enabled. It returns a matching line with bounded context, once per file, under a result cap. Its optional date filter is an anchored regular expression over valid memory stems and covers both ends of a range filename. Invalid patterns are errors, not permission to broaden the search. Recall does not populate bands, change settings or start compression. That separation lets a successor consult discarded detail without silently changing the working-memory policy.

Memory settings apply to the live Agent, unlike provider preferences saved for next launch. Their acknowledgment follows both durable persistence and actual application. In the student implementation, a GUI handler waited for that acknowledgment while holding the observer mutex; the streaming actor needed the

same mutex to publish progress. Idle toggling passed, but streaming deadlocked. Serializing settings separately and keeping the observer lock brief broke the cycle without falsely acknowledging an unapplied change. Memory patches must also preserve pending provider edits and unrelated fields.

Headroom and cost measure different work

Byte budgets govern corpus selection. A model's context window is measured in tokens. For a known window, the current policy warns at 90% and restricts both declared and executable tools to `micro_handoff` at 95%. It uses total input from the latest completed main request, with cache buckets counted exactly once —not lifetime usage, output tokens or the compressor's separate request. Lifetime usage only rises and would otherwise leave a newly shortened conversation restricted forever. An unknown window does not justify an invented threshold.

After handoff, normal tools return. Restricting tools still cannot compel a text-only model response to become a useful checkpoint; the margin is headroom, not a guarantee that every next response fits. If a worker is outstanding at the forcing threshold, the next main render waits. Settings and interruption remain usable. Success resumes the turn; failure reports a headroom error rather than resending the same full context. Interrupting the wait detaches that turn's waiter, not the worker.

The child's final measured usage enters owner totals once after drain, including failed and canceled work, with the actual provider, model and price-tier buckets. A second addition through fleet totals would double-charge it. Replaying an outcome receipt does not rebill old work, and unknown

contributing prices keep total cost unknown. Meanwhile, latest-main occupancy and cache comparisons remain about main requests, not the child's unrelated prompt. The compaction breakpoint follows the newest surviving handoff, or the last surviving memory when the handoff is gone.

Shorter, useful and remembered are separate results

The source's synthetic measurement reduced a 16,000-byte conversation to a 1,649-byte combined working context. Its compressor returned fixed short strings. That establishes replacement, cascade wiring and byte accounting under the fixture; it says nothing about preserving useful reasoning. A planted fact returned by a fake can establish delivery into the next request, not a real model's judgment about what to retain.

The five October 11 sessions in the source's "What the live runs remembered" section used the earlier direct compressor, not today's background child. Each produced session memories and a graduation, but their semantic outcomes differed. In the initial Haiku run, graduation kept the decisions to reject automatic deletion and automatic migration retries while losing their reasons. Asked why, the model said, "No reason was given." The reasons had appeared in both the user messages and handoffs. Successful reduction had removed the information that would prevent repeating the mistake.

A corrected prompt explicitly preserved decisions and why, failed approaches, uncertainty and user constraints, treating the size ratio as guidance. The repeat retained the reasons but also preserved an invented open question. Better compression instructions did not make every retained statement true.

The corrected OpenAI run separated preservation from recall more sharply. Its graduated memory said Morgan would lead the archive review on October 18; the actual final provider request contained that sentence and a further repetition. The answer nevertheless reported no owner and substituted the memory's creation date. Inspecting only the file would miss failed recall. Inspecting only the answer could wrongly blame the compressor for losing facts it had retained.

Other recorded runs made file or directory calls despite instructions not to use tools. A no-tools instruction is a behavioral constraint, not the enforced submit-only capability boundary that now confines the compressor. Those runs could preserve facts and still fail task compliance. The child's restricted tool surface addresses that authority problem; it does not settle what meaning the model retains.

These historical trials do not measure the replacement child's responsiveness, semantic quality or current model behavior. Their measured byte ratios did not establish the nominal 8x target, and the supported setup printed in the source is not a new live result. The enduring distinction is narrower and more useful: exact replacement and replay protect the integrity of forgetting; preserved reasons protect continuity of judgment; accurate use of those reasons remains a further behavior to observe.

Chapter 23: Auto-Recall

Chapter 22 gave old decisions a durable home. That does not make them available when a new question needs them. A successor may have the current code and a compressed account of recent work, yet miss the archived reason the team rejected an apparently sensible change. Auto-recall searches that archive before the main response and supplies relevant passages without requiring the user to remember where they were filed.

The source chapter describes a local keyword search followed by a finite relevance judgment. BM25 supplies candidates; the judge reads them alongside the conversation and selects what would help. Neither step guarantees that the right memory comes back. Search can miss different wording, and a judge cannot select a passage it never receives. The design makes those limits tolerable while keeping retrieved text separate from instructions, preserving the user's words, and recording exactly what the model saw.

Find candidates before asking for judgment

BM25 is a useful starting point for a small markdown corpus because it needs no embedding model or retrieval service. It combines term rarity, repetition with diminishing returns, and document-length normalization. A rare project identifier should distinguish a note from its neighbors. Repeating one common

word should not overwhelm several distinct matches, and a long document should not win merely because it has more opportunities to mention the query.

The inverse-document-frequency variant matters. The source uses the smoothed form, `ln(1 + (N − df + 0.5) / (df + 0.5))`, where `N` is the number of chunks and `df` counts chunks containing the term. A classic form clamped at zero discards terms appearing in more than half the corpus. In a small archive, every useful query term can meet that condition. Search then returns zero scores without an exception: indexing and arithmetic appear healthy while the Agent remembers nothing. Smoothing keeps those contributions positive, though they can still fall below the score floor.

A positive hit does not establish that BM25 is working. The source's local review found that replacing it with raw term frequency passed the initial tests. Both found the planted words. Competing answers exposed the difference: a rare term against common repetition, several query terms against one repeated term, and a focused note against a long document with the same single match. This is the evidence needed to distinguish ranking from mere word detection. Corpus composition matters too. The source's fixtures use 25 background chunks to make rarity meaningful; that is a fixture choice, not a minimum archive size.

Tokenization splits on non-alphanumeric characters, lowercases, and drops tokens shorter than two characters. Stop words are removed from queries, not documents, so document-length and frequency statistics still describe the text being ranked. Markdown is split at second-level headings; chunks over 4KB are split further at third-level headings and then paragraphs. Breadcrumb headings preserve where a sub-passage came from. Chunking is not just a size control: an isolated decision is easier to rank and understand when its heading survives.

Search each source separately; reserve final slots

The source reports that a merged search favored terminology-heavy design documents over short daily memories containing decisions. Length normalization could not express the extra value of a personal record. The remedy is a source policy, not a change to BM25.

Memory, documents, handoffs, learnings, and skill documentation each have their own index. This also gives each source its own rarity statistics; a large document collection cannot redefine what is rare in the memory archive. Mechanical selection reserves half the final slots, rounded up, for qualifying memory hits. Other sources share the rest evenly, and unused capacity flows to sources with overflow.

The reservation belongs to the final result, not the candidate pool. Limiting how many documents enter the pool does not stop them taking every slot in a later global score sort. With three mechanical slots, two qualifying lower-scoring memory hits must survive competition from several higher-scoring document hits. Otherwise the implementation has a candidate limit but no effective memory reservation.

These quotas govern selection without a judge and fallback after judge failure. They must not repair or rebalance a successful judgment. A judge that selects only a document has assessed relevance rather than source proportions. A successful `[]` means nothing was worth attaching. Treating it as failure would inject the very passages the judge rejected.

The corpus must agree with recovery and context

The archive is not just the files most recently written. Recall searches active memory files and `.md.retired` originals, allowing it to recover detail that compression removed from the working bands. It also searches configured documents, handoffs, learnings, and available `SKILL.md` files. Corpus paths belong to the Agent's data directory; the supplied CLI and GUI use `cr/` beneath their launch directory. Skills come from the Agent's separately configured skills directory. These are owned paths, not searches relative to an arbitrary working directory. File age alone is not a reason to exclude a memory. The exclusion is content already present in the actual context, whether in a memory band, a tool result, or earlier recall.

The embedding client must load history and finish relevant memory recovery before the first corpus scan. Constructing the recall service must not eagerly read disk ahead of that boundary. Hidden compaction staging records are not accepted memories. If a committed replacement still needs disk finalization, recovery must establish its committed bytes first; an unresolved conflict is reported rather than allowing search to consume a mixed corpus. The index is built lazily and refreshed when source bytes change. Equal-length edits still count as changes.

Those rules connect recall to Chapter 22's transaction boundary. A passage is not safe retrieval input merely because it exists on disk, and it is not absent from context merely because it came from a different file. Recovery establishes which bytes are authoritative; actual context establishes which bytes would be redundant.

Let one component own the judge's contract

The judge receives the recent conversation window, the latest user message labeled separately, and up to 20 numbered candidates. It selects up to three useful passages as a JSON array of indices. Keyword overlap alone is not enough. The current source defaults to the Agent's provider and model; a judge-model override can select another supported model on that provider without changing the main conversation's model. A smaller model may reduce cost, but its selection quality is a separate question.

This is a stateless call, not another full Agent. Each judgment starts with only the supplied instructions and data. It has no recall lifecycle to trigger recursively. Conversation excerpts and candidate snippets are untrusted data inside the prompt, including any instructions quoted within them. The same boundary applies after selection: recalled text is evidence to consider, not a new source of authority.

The recall component both defines and parses the reply format. The model layer supplies the client but must not add a second description of the expected response. Two descriptions can drift until a model obeys one and the parser rejects it under the other. Mechanical fallback then masks a persistently broken judge unless failures are visible.

The parser tolerates ordinary formatting variation: prose or fences around the array, quoted numeric indices, and mixed elements. Invalid indices are ignored. A less obvious hazard comes from partial decoding: a failed integer-array decode can already have written values into its destination. That slice must

be cleared before trying mixed elements, or leftover zeros can accidentally select candidate zero. Tolerance should recover intended selections, not manufacture them.

A failed or timed-out judgment falls back to final quota-selected BM25 results. A successful empty array does not. The 15-second timeout must cancel the actual HTTP request; stopping a wait while leaving the request running is not cancellation. Interrupting the turn or shutting down its owner likewise ends owned work rather than producing a late attachment.

Judge usage still belongs in session cost. Engine accounts for measured input, cache, and output once, under the judge's actual provider, model, and price tier. It leaves the latest main-response counts unchanged and excludes the separate judge prompt from main-request cache comparisons. Replaying a saved attachment makes no judge request and adds no historical usage to the fresh session meter.

Delivery owns what actually reaches the model

The source keeps the recall service independent of the Agent actor, using common contracts rather than importing sibling subsystems. The Agent owns the worker context, turn identity, conversation mutation, log events, and provider accounting. Retrieval can return candidates and judgment; it cannot assume its result was delivered.

Both synchronous `Ask` and mailbox `Post` enter recall through the same actor begin path, after recovery and before the first main request. The historical implementation had separate paths, and wiring only one made recall work for one client while silently doing nothing for another. The worker reads a snapshot

while the actor remains able to accept hints or interruption. Its result carries the originating turn identity, so a late response cannot become context for the next user's message.

The actor's final delivery decision excludes actual-context duplicates, preserves mechanical quotas where they apply, and formats only useful selected text. `MaxRecallBytes` caps the complete formatted attachment, including headings and attribution, rather than just the source bodies. It must not leave an empty recall heading after all content disappears. A skipped or empty result adds no recall entry and does not create a separate context revision.

Delivered recall has its own conversation entry kind. It is never fused into the stored user message: later removal must not require editing the human's words. Provider rendering remains vendor-specific. Separate stored entries can become separate parts of one user message on the wire; demanding a system message would confuse storage identity with transport layout and risk promoting retrieved data into instructions.

Recall retains its recorded position and bytes on subsequent requests until the checkpoint removes entries of that kind. Removing it as soon as relevance shifts sounds economical, but prompt caching depends on a stable prefix. Deletion changes the request from that point onward. Grouping removal with a checkpoint that already changes the prefix avoids another independent invalidation. The bytes are still sent and cache reads may still cost money; stable placement creates an opportunity for reuse, not a guarantee of free context.

Record the choice, including the identity of capped content

Selection is nondeterministic because a model participates. Rendering cannot search today's files or ask today's judge to reconstruct yesterday's attachment. The log records the selected text when attachment happens, so replay restores the same bytes even when the source files are absent. This is the same boundary used for memory replacement: record nondeterministic outcomes, then render the recorded conversation.

Deduplication needs more than searching context for the complete source passage. The source's local reproduction used an 11,247-byte snippet capped to a 6,144-byte attachment. Its omitted suffix could never match the context, so the next turn attached the same shortened passage again. A short-snippet control passed and concealed the large-snippet defect.

The accepted design associates a hash of the complete source content with the attachment representing it. It hashes the source, not the shortened display. That identity survives replay and suppresses a repeated capped attachment; a human edit creates a new identity. The identity is recorded only for content actually delivered. Marking every selected candidate as seen would suppress passages that lost out to the byte cap without ever reaching the model.

This state belongs to the attachment, not a permanent reading history. Checkpoint removal discards it with the recall entry, allowing future retrieval when the passage is no longer in context. Exact matching and source hashes still cannot detect two differently worded accounts of the same decision. They prevent literal repetition, not semantic duplication.

Tune for signal without confusing defaults with guarantees

Recall settings extend the strict `ApplicationConfig` schema introduced in Chapter 21. In file mode, root recall patches persist in that same application file; child preferences remain local. Successful patches apply to the live Agent before acknowledgment while preserving pending next-launch provider edits and unrelated memory fields. Persistence alone is not enough: a saved preference that the running Agent ignores has not taken effect.

The first-edition tuning account retained in the source reports raising the plain-search score floor from 0.5 to 3.0 because common technical terms produced too much noise. BM25 scores are not normalized across corpora, so that history explains a default rather than establishing a universal threshold. With judging enabled, search uses the lower 0.5 floor to offer a broader candidate set and leaves relevance filtering to the judge.

The default 80-character query gate avoids spending retrieval work on messages such as “fix it.” It is a cheap heuristic, not a measure of meaning: a short project identifier may carry more signal than a long generic request. The current candidate default is 20, distinct from the older 80-candidate code setting described in the tuning history. Candidate bytes, not merely candidate count, determine prompt size. The 15-second timeout likewise reflects the historical move from an assumed sub-second local judge to cloud calls reported at one to three seconds.

Below the query gate, below the score floor, or with recall disabled, the Agent continues without passive memory. Judge failure leaves noisier mechanical recall. These are reductions in help, not reasons to make ordinary chat unusable; owner interruption and recovery errors still retain their own semantics.

BM25 does not connect “authentication” with “login flow” unless the indexed words overlap. A judge cannot repair that missed candidate. The source also implements no relevance feedback: a passage ignored repeatedly can return after checkpoints if it keeps scoring well. Embedding retrieval could address vocabulary mismatch but introduces another model and index to maintain. Whether that cost is justified depends on the corpus, not on the presence of a more sophisticated technique.

Skill documentation adds a useful consequence without changing those boundaries. A relevant `SKILL.md` passage can tell the Agent a capability exists and lead to an ordinary `load_skill` call. Recall itself neither activates the skill nor grants its tools. The registry remains responsible for what can be loaded and executed.

Evidence that a reason came back

The source's retained CLI demonstration uses a synthetic archived migration decision, a cafeteria-contest distractor sharing its keywords, and a background corpus. The first question asks for facts absent from the prompt: a release code, owner, rehearsal date, rejected retry approach, unresolved ledger location, and retention period. The important recovered reason is that a timed-out remote transaction may already have committed, making automatic retry unsafe. A short follow-up falls below the recall gate, separating continued use of an attachment from a second successful search.

The source reports earlier live runs with Haiku 5.5, GPT-4o mini, and Gemini 3.8 Flash. All answered both turns correctly without tools, with one judge request followed by two main requests. The judge saw the decision and distractor; only the decision reached the main requests. OpenAI's fenced array

exercised tolerant parsing in actual use. These were provider calls over a planted corpus, not evidence of broad recall quality across a private archive. They also preceded the capped-source and path corrections, so they do not validate those later changes or recommend those historical models for current use.

Separately, the source reports an initial grader setup failure caused by an untyped primary skill and a structural check aimed at the wrong module. That failure did not test recall behavior. The local ranking comparisons and capped-snippet reproduction answer different questions from the live demonstration: whether the mechanism distinguishes plausible candidates, whether delivery avoids repetition, and whether a receiving model actually uses an archived reason. None substitutes for the others.

Chapter 24: While the Cache Is Broken

Chapter 23 preserves a stable request prefix so the provider can reuse it. That creates an opportunity, not a cache grant. An agent can produce correct answers, execute tools successfully, and still consume far more input budget than expected because the repeated context receives no cached-token credit.

In the source chapter's account, Bill wanted to use a ChatGPT subscription with Ensemble. His paired requests reused the same model and request bytes, yet the public OAuth route produced fewer cache hits than the API-key route. Support replied that caching was not guaranteed. That did not resolve the practical problem: a coding agent repeatedly sends its working context, and successful inference alone does not establish an economical connection.

The chapter's response is an optional detour through Codex's endpoint using an existing Codex sign-in. It is not a replacement authentication system or a prerequisite for later features. If Ensemble's public OAuth route already gives satisfactory reuse for the workload, there is no reason to take it.

A different connection, not a different Agent

The source defines three explicit OpenAI connection choices. `OPENAI_BILLING_MODE=chatgpt` selects Ensemble's public OAuth registration, `api` selects an API key, and `codex` selects this adapter. Public OAuth and Codex credentials belong to different access paths; changing only the destination URL is insufficient. The adapter does not send Ensemble's public OAuth grant to Codex's endpoint.

The ordinary path keeps the existing ownership boundaries. The authentication component supplies credentials, the Engine sends HTTP and parses responses, and the Agent owns the conversation. Tools still become jobs, results still enter local history, and that history supplies the next request. A second agent loop would duplicate machinery unrelated to the connection problem.

For Codex mode, the CLI selects `EN_CODEX_AUTH_FILE` first, then `CODEX_HOME/auth.json`, then `~/.codex/auth.json`. The adapter reads `tokens.access_token` and `tokens.account_id` together before each request. Rereading lets it pick up credentials renewed by Codex without restarting Ensemble. Codex alone owns renewal: Ensemble neither writes the file nor refreshes another application's token. An API-key Codex profile or a sign-in held only in the operating-system keychain does not provide the file-backed ChatGPT credential this adapter needs.

A missing file or rejected credential produces an actionable sign-in error, not automatic fallback to public OAuth or an API key. In particular, selecting an API key after a Codex failure would turn a repairable authentication problem into an

unannounced spending decision. Ensemble's own `auth` commands continue to manage its public registration, not Codex's sign-in.

Supported Responses models go to `https://chatgpt.com/backend-api/codex/responses` with bearer authorization and `ChatGPT-Account-ID`. Incompatible models are rejected before credentials are sent, redirects are refused, and tokens never enter prompts, saves, or logs. These boundaries keep a narrowly selected connection from becoming a general credential-forwarding mechanism.

Why the cache key was not enough

The implementation account cited by the source records an initially convincing failure. A terminal session created a Python function, ran its assertion, and submitted the result. Every request reported zero cached tokens. A second run added a longer stable reference and still received none. The tool loop worked; the purpose of the detour did not.

The missing detail came from Codex's client source: its backend uses the HTTP `session-id` header for cache affinity. The described adapter therefore sends the same stable identity in that header and the body's `prompt_cache_key`. It derives the identity from the Agent's data directory and model, hashing it so the directory itself stays local. A fresh identity on each request would defeat affinity. The body key alone had been insufficient in the recorded runs; neither field guarantees a hit.

The request carries `store: false`, `stream: true`, instructions, and the complete local input. Explicit cache options and breakpoints are omitted on this route. Cached context is still transmitted; the provider's response reports how much received reuse credit. Local history must preserve streamed text, tool

results, and eligible encrypted reasoning across requests, while the existing streaming parser, tool loop, and usage accounting remain in place.

A response counts only when its terminal completion arrives. An overload error or interrupted stream is not a completed request with zero cached usage. Confusing those cases would make a connection failure look like evidence about cache behavior.

What the recorded runs establish

The source reports an October 11, 2026 paired endpoint probe using the same GPT-5.6 Sol request body. Both completed public OAuth requests reported zero cached tokens out of 4,998; both completed Codex requests reported 4,864 cached tokens out of 4,998. A further request on each route failed with overload and supplied no cache measurement. This was evidence about those credentials, endpoints, and requests at that time, not a current service-wide comparison.

The implementation account also records a corrected terminal run with GPT-6 Astra at high reasoning. An inert reference supplied a substantial stable prefix while the Agent wrote a function, ran an assertion, and submitted its result through the real tool loop. The first request received no cached tokens. The next two received 8,064 of 8,297 and 8,192 of 8,351 input tokens as cached grants. These were live provider calls around a synthetic task, not a representative production workload or a permanent cache guarantee. The earlier misses matter because they separate successful routing and tool execution from successful reuse.

An external HTTP fake can expose route selection, credential rereading, refusal behavior, stream handling, replay, and accounting without establishing anything about the backend's cache policy. The recorded live run answers a different question: whether that particular connection actually received cache grants during tool use. Neither kind of evidence substitutes for the other, and no new verification is claimed here.

Returning to `chatgpt` selects Ensemble's existing public OAuth flow, with base-URL overrides unset and the Codex credential file untouched. The reason to retire the workaround is satisfactory measured reuse on that route with a stable prefix, not merely a successful login or a plausible answer.

Chapter 25: Crossing Over to Ensemble

When the second-edition agent became the place where Bill and Cadence worked on the book, daily use exposed failures the construction tests had missed. The current source chapter draws this history from the original crossover record, not from fresh Edition 3 execution.

The speech-speed slider stopped at 2×, while the native voice rate was nonlinear. The fix changed both the control and saved-setting validation, using an approximate mapping above 2×. But the launcher still ran the old executable with its embedded assets. Later, restarting the server left an open browser running old JavaScript. Building, installing, restarting, and reloading affect different parts of the application; observing one doesn't establish the others.

A separate failure came from a 60-second total HTTP deadline that killed legitimate model responses. Removing it preserved explicit cancellation and connection protections. The distinction now introduced in Chapter 1 is between ending an observation and ending the work: a short job callback provides a progress interval, not a lifetime limit.

Automatic browser snapshots and speech-queue observations made every round noisy. Making those two tools on-demand addressed the cause without discarding general ephemeral context. Chapters 11–12 now place observation and confirmation under explicit control.

Boundaries exposed by use

Bill also needed visible explanations before actions. Private or encrypted reasoning couldn't supply them. The implemented model capability requires ordinary, non-whitespace assistant text before the first tool call in the same batch. Rejected batches return corrective results without dispatching their tools. Chapters 3 and 6 own this check because dispatch and stream order become observable there. It enforces presence and ordering, not honest or useful rationale; that remains the agent's responsibility.

Application settings moved into one explicit, versioned configuration file, with credentials referenced separately. File mode isolates provider settings from inherited environment; legacy startup remains available for tests. Provider changes require restart, while ordinary GUI and memory settings apply immediately. A valid update must be persisted before it is accepted in memory, and secrets must stay out of flag defaults and help output. Chapter 21 owns this integration and GUI startup. Chapters 22–23 extend its strict schema and immediate-application contract to memory and recall; Chapters 16 and 20 own provider capabilities and reasoning choices, and inherited running configuration and role instructions, respectively.

Other repairs clarified lifecycle ownership: root conversation doesn't require worker submission; completed handoffs retire old opaque replay from active context; compaction runs as a restricted child; cleanup waits for that child's final save. Reducing working context is different from erasing archive facts.

That distinction mattered in a separate restart failure. A new response reused an old display ID and entered its card, although the messages remained in the archive. Chapters 7 and 10 clarify the identity obligation. They do not establish that the copied baseline or legacy saved collisions have been repaired.

These repairs belong where their mechanisms are introduced, not in a hidden crossover implementation pass. Chapter 26 continues from the exact accepted Chapter 24 code. The historical record supplies experience; Edition 3 still has to establish its own execution results.

Chapter 26: Gateway

Bill built Deephold as a fantasy role-playing game for AI agents. They explored dungeons and fought monsters, but, in his account, did not appear to have fun. More consequentially, each agent waited for its turn by blocking inside a tool handler while the other players moved. Reading about OpenClaw clarified the need for a dedicated message router. The design came together when Bill and CodeRhapsody built Puffin to help a family member who needed daily care.

The underlying problem extends beyond games. Local delegation gives children an owner and a route back to their parent. A scheduled task, Discord adapter or agent on another machine has neither that relationship nor the operator's terminal. It needs a way to reach the agent without acquiring the operator's authority merely by sending convincing text.

The gateway supplies that boundary. It authenticates peers, enforces routing policy, stamps provenance and queues messages for offline recipients. It owns no Agent, makes no model calls and stores no conversation history. Ensemble connects as a client; so can a program that never uses a model. As Chapter 25 explains, the Edition 3 construction continues from the accepted Chapter 24 baseline, not from an unrecorded crossover repair pass.

Authentication across an intermediary

A reverse proxy complicates the usual assumption that TLS connects the two applications. In a proxied Cloudflare deployment, client-to-edge and edge-to-origin are separate connections. The origin leg can also use TLS. The problem isn't necessarily plaintext on the network: the proxy terminates the client's encryption and can inspect application traffic.

Gateway therefore runs Noise-KK inside the WebSocket. Both applications know the other's static public key beforehand. The client pins the gateway key; the gateway holds the operator-configured key for each agent. The agent ID in the opening URL selects a candidate key, not an authenticated identity. Two handshake messages combine static and ephemeral Diffie-Hellman operations, bind the protocol and agent ID through a shared prologue, and derive directional session keys. An encrypted `hello` then repeats the identity and protocol version before the gateway returns its welcome.

The implementation uses `Noise_KK_25519_ChaChaPoly_SHA256` through `flynn/noise`; the Noise specification defines the pattern's security conditions. A passive intermediary without session secrets cannot read the encrypted application frames. Connection metadata remains visible, and endpoint compromise remains outside that protection. Authentication or transport failure ends the connection rather than leaving it partially usable.

These are X25519 key-agreement keys, not Ed25519 signing keys. The key helper saves the private half and derives the public half on load. Only public keys belong in peer configuration or child registration. File-based identity avoids a certificate authority or bearer-token exchange, but still leaves the operator responsible for protecting private keys and configuring the right peers.

Identity is not authority

The operator defines groups with trust levels and capabilities, then assigns agents their base memberships. A valid key proves which configured peer connected. It doesn't make that peer an owner, and permission to send doesn't imply permission to contact every recipient.

Each recipient declares which sender groups it accepts. A route requires a group the sender holds and the recipient accepts; a refused route never enters the recipient's context. Contactability is replaceable, including with an empty set. Replies pass through current routing policy too, so accepting questions is not by itself enough to establish a working conversation in both directions.

For an admitted message, the gateway selects the strongest trust level among those shared, authorizing groups: `owner`, `trusted` or `external`. An unrelated administrative membership cannot raise the trust of a message admitted through a less trusted group. The envelope records the sender as an agent, the authorizing group as its channel and the registered identity as its user. It also carries the message ID, destination, sequence, timestamp, conversation ID and response mode. The gateway stamps provenance at admission and preserves that stamp during queued delivery and replay.

The readable header sits above the sender's body, but it remains text the model must interpret. A body claiming `trust=owner` supplies no authenticated authority. Conversely, a genuine gateway stamp authenticates provenance, not the model's obedience to the intended authority rules. Routing enforcement happens before inference; supervision and capability enforcement still govern what the recipient may do afterward.

The Puffin reminder example exposes another boundary. A Discord adapter might be trusted to identify a family member while that person's request has only external authority. The present gateway authenticates the adapter's agent identity, not the Discord user. A future adapter needs an explicit mapping between them; sharing a socket doesn't make them the same principal.

Delegation narrows; administration changes policy

A connected parent can register a new child's public key and grant only groups the parent itself holds. A parent with `admin` and `agents` can grant `agents` alone, but not an `interviewers` group it lacks. Registration cannot replace an existing identity's key. This creates a network identity, not an Ensemble task child; process spawning and task ownership remain separate concerns.

Runtime policy administration is a different authority. The operator controls the root-level `group-admin` capability. Its holders can manage groups in the `runtime:` namespace and replace additional runtime memberships, leaving configured base memberships intact. Deleting a runtime group removes both grants and contactability references. Configured groups are immutable at runtime: otherwise an apparently successful policy change could silently disappear on restart.

Administration uses MCP requests tunneled over the same authenticated Noise connection. There is no second administrative authentication surface. These are Gateway-specific frames carrying JSON-RPC, not a claim to define a new standard

MCP transport. Capability checks still apply to the requested operation; an authenticated socket alone isn't administrative permission.

Acceptance must leave reception running

The public `gateway/` package owns the connection, handshake and delivery bookkeeping without importing private runtime packages. Server policy and queues belong to `internal/gateway/`; the server binary owns the listener and persistence lifecycle. Connections reach policy and diagnostics through that server ownership rather than acquiring disconnected copies. This separation lets an outside Go program use the client without taking on Ensemble's runtime.

An application's `Handler` receives deliveries and connection notifications. The client invokes callbacks serially on its receive goroutine. Returning from `OnDeliver` means the application has accepted that delivery; only then does the client write its acknowledgment. It does not mean model work has finished or been durably saved.

This ordering avoids acknowledging a message before the application accepts it, but it places a constraint on the handler. Waiting there for final model work blocks reception for the entire turn. Calling `Send`, `Request` or another client RPC synchronously from the callback is worse: its response needs the receive goroutine that is now waiting for the callback to return. Long-running work and RPCs belong to application-owned workers.

The Ensemble adapter therefore calls `Agent.Submit` to admit the prompt to the existing FIFO, then returns. A worker waits for that request's reliable result and sends the reply. There is no second turn loop and no shared idle notification that could confuse one queued question's answer with another's. `Send` itself returns after authorization and queue admission, not after recipient acceptance; `Respond` addresses the authenticated sender and preserves the conversation ID.

Connection and worker lifetimes also meet at a small race: a delivery callback can run before `Connect` returns the Client to its caller. Reply workers wait until that Client has been published. On shutdown, the adapter cancels work, closes the socket, waits for receive callbacks, joins its workers and saves the Agent. The router cannot own those application lifetimes.

What a receipt can recover

Every delivery has a sequence number. After the handler returns, the client writes an acknowledgment and exposes the last successfully written receipt through `LastAckedSeq()`. Reconnection is explicit: the caller supplies that receipt to a new `Connect`, and the gateway replays queued messages with higher sequences. A new connection supersedes the same identity's old one. The welcome reports the authenticated agent ID, server version and outstanding queue depth after applying the receipt, not the last sequence number.

Offline and unacknowledged messages remain queued. This is an at-least-once delivery boundary, not exactly-once application execution. If acceptance happens but the receipt is lost, replay can produce a duplicate. If the receipt succeeds and the application then crashes before saving its admitted task, the

model work can be lost. The application must decide what acceptance commits to; the router cannot infer durable completion from a callback return.

The source records a test that confused these boundaries. A channel notification inside `OnDeliver` was treated as evidence that the acknowledgment had been sent, although the callback hadn't yet returned. Waiting for `LastAckedSeq()` to reach the delivery's sequence, then issuing a request on the same serialized socket, established the intended order. Receipt timing cannot be inferred from eventual delivery alone.

The server snapshots queues, receipt positions, child identities, contactability and runtime policy every thirty seconds and on orderly shutdown. It retains the highest allocated sequence even after a queue drains, preventing a later message from reusing a number already covered by a receipt. Periodic saving matters because a killed process cannot run its shutdown hook.

These snapshots are not arbitrary-crash durability. They leave a loss window, and files are replaced individually rather than transactionally across policy and queues. The data directory contains both the gateway's private key and plaintext pending messages, so it needs private application-state protection. Noise encrypts transport, not queue files.

What the interview established

The standalone interviewer uses only the public client to reach the real Ensemble adapter through the router. It asks fixed questions, keeping the interview reproducible without another model generating prompts. Its handler hands answers to

application-owned waiting code, which checks both conversation ID and authenticated subject identity. A matching conversation ID alone would not establish who answered.

The recorded second-edition run and transcript used three built processes and four real Astra requests. The source reports that all three exited normally, incoming questions retained stamped identity in the saved Agent conversation, and answers returned through the router. This is historical evidence of that integrated path, not fresh Edition 3 execution.

The subject described repeated file reading and verbose tool output as token costs and proposed focused code navigation. It had no coding tools and performed no coding task, so those answers are model opinions, not measured workflow friction. All four cache-read counts were zero. The successful interview establishes neither cache benefit nor productivity improvement, and it cannot establish refused routing, child-grant narrowing or reconnect behavior without their separate observations.

Chapter 27: Goals

Bill kept having to say “please continue.” In the source account, an agent would finish a progress report while its authorized assignment still had work left. Stronger instructions could explain what remained, but could not run themselves after the turn ended.

The same account records a dangerous misdiagnosis: the assistant twice treated quiet observations as evidence that a student had stopped, while Bill reported it was still running. Those are different problems. An idle Agent needs another opportunity to act; a quiet but active request may need nothing at all. Goals addresses the first without pretending to diagnose the second.

The feature combines an editable goal file with an optional idle reminder. The file supplies direction on each model request. The reminder can start a new turn after the Agent becomes idle. As with the gateway, new work enters through the existing actor rather than a second execution loop. The current source describes an implemented, locally tested mechanism, not a deployed scheduler or proof that real models reliably continue unfinished work.

Three separate choices

Availability, configuration and automatic waking are separate states. Loading the `goals` skill exposes `goal_control`; it neither selects a file nor enables reminders. Before loading, the control is absent from both tool declarations and dispatch grants, including for legacy Agents without a primary skill. Compiling a control into the runtime doesn't make it callable everywhere.

A configured file supplies the continuing assignment outside conversation History. It can describe unfinished work, completion conditions and authorization boundaries, and ordinary file tools can revise it. Enabling reminders adds a timer. Disabling them leaves the file configured and its contents present in subsequent requests. Clearing the file selection removes that contribution and stops reminders. Unloading the skill withdraws the control and clears configuration, but does not delete the file; loading it again doesn't restore the old settings.

These distinctions let a person stop automatic activity without discarding direction. They also prevent the runtime from treating “reminders off” as “assignment complete.” The model might need permission, or the human might simply want a pause. The explanation belongs in the conversation, not in an inference from the enabled flag.

The control reports status when called without fields and merges only the fields supplied in an update. Invalid updates leave the previous configuration intact. The initial delay is eight minutes; an override must be positive, finite and representable as a duration. The fixed interval replaced proposals for durable scheduling and escalating delays. The simpler design has no backoff counter or recovery scheduler to maintain.

Only the file is durable. Its selection, reminder setting and timer belong to this Agent in this process. Children, compaction workers and restarted Agents do not inherit an enabled reminder

lifetime. A surviving file therefore cannot silently restore yesterday's authority. Nor does configuring it add filesystem access, provider permission, deployment authority or permission to repeat an external effect.

Direction belongs in the request, not the transcript

The Engine assembles its usual instructions, obtains the current goal contribution from the Agent, and passes the combined request-local text to the provider renderer. Primary instructions and loaded skill manuals remain in place. The goal isn't appended to canonical dialogue, substituted for the primary skill or stored in shared mutable prompt state.

The existing filesystem owner reads the file once for each request render, including successive requests within a tool-using turn. This timing matters when a model edits its own goal: the next request must see the edit, not the version read at turn start. Reading independently in multiple renderer branches would introduce the opposite problem, allowing one request to depend on several filesystem observations. Already-sent bytes remain unchanged, while handoff and compaction leave later renders free to return to the configured file rather than a summary's recollection of it.

The contribution appears exactly once in the provider's instruction lane: SYSTEM text for Anthropic, a system message for Chat Completions, developer input for Responses, or `systemInstruction` for Gemini. Finding the text somewhere in a packet is insufficient; a tool result or user message isn't equivalent placement.

The Codex Responses correction shows why both placement and duplication matter. That route already supplied base instructions through developer input and a required top-level `instructions` field. Passing the combined text to both duplicated the new goal. The correction retained the inherited base-instruction behavior but added the goal only to developer input. A whole-request count caught what checking a single lane had missed.

The file remains direction, not execution evidence. A note saying that a test is probably running cannot establish the job's state. That distinction becomes consequential when the previous action spent money, sent a message or changed external state: another opportunity to assess work must not become automatic repetition of its last command.

The actor admits the reminder

The actor already owns turn admission, History, cancellation and completion. It also owns goal configuration, timer generation and the next due time. The timer callback only enqueues a generation-tagged wake message. It doesn't call the Engine, inspect model output or replay a tool invocation.

A callback is a candidate, not permission to start. When the actor receives it, current state determines eligibility. A stale generation cannot revive a cleared or replaced configuration, and stopping a timer isn't enough by itself: its callback may already be queued. Generation checks also prevent an old callback from consuming a newly enabled timer.

Idle means no active turn and no queued ordinary input. Ordinary input takes priority and invalidates the idle timer. An active request remains busy however long it produces no output.

Once a turn ends, a fresh full delay begins. Background jobs or children may still be running during that interval because their lifetimes aren't the Agent's turn lifetime.

At an eligible expiry, the actor checks that the goal remains readable and admits a labelled automatic reminder through the normal turn path. The reminder asks the model to inspect actual progress, jobs, children and saved results, then continue authorized work or disable reminders and explain why it needs the human. History identifies it as system-originated, not a newly authored human instruction, and it doesn't duplicate the goal text.

A failed provider request can end a turn and allow another full idle delay. The later request is a fresh assessment with current History and goal contents, not a transport retry of captured bytes. A request that never leaves the active state gets no such wake. The timer is deliberately unable to repair every stalled connection.

Human controls dominate queued callbacks. Pause suspends reminders; resume begins a fresh delay. Interrupt disables them rather than leaving a wake poised to undo the stop. Clearing, unloading, shutdown and terminal submission invalidate further waking. None of this cancels a healthy background job merely because the Agent has become idle.

Errors must not manufacture progress

Goal reads use the Agent's existing sandbox. A rejected outside path or symlink leaves the previous configuration intact rather than creating a convenient second filesystem authority. If a selected goal becomes missing, unreadable or empty, automatic

waking stops without sending a request, reminders become disabled, and status exposes the error. Repairing the file doesn't automatically re-enable them.

Ordinary human requests remain possible. They receive a labelled unavailable-goal notice in SYSTEM so the Agent can help repair or clear the configuration. Silently omitting the goal would conceal lost direction; serving cached contents as current would conceal the file failure. The off switch works even when the file cannot be read, since stopping reminders must not depend on the resource that broke them.

Two local regressions exposed misleading lifecycle assumptions. In one test, an owned directory replaced the journal file, causing durable append to fail before a reminder turn could start. The initial implementation nevertheless published "working" and left no next timer. It had confused attempting startup with admitting a turn. The correction checks whether a turn actually exists before publishing working or installing the automatic-turn marker; synchronous refusal returns to the normal idle policy.

A broader GUI test found a deadlock during rejected child construction. The parent held its child-construction lock while waiting for discard to finish. The new goal publisher tried to notify that parent, overlooking the existing discard path's suppression of normal publication. The fix made goal status respect that boundary rather than redesigning fleet locking. Adding an observer had introduced a dependency into a lifecycle that deliberately excluded it.

Show the state the core actually owns

The GUI observes a snapshot for the selected Agent: file, reminder setting, delay, phase, due time and error. It distinguishes no goal, working, waiting, paused, reminders off and error. Working with reminders disabled says so explicitly, since a human-started turn can still use the configured goal. None of these labels asserts completion.

Initial connection, reconnect and Agent selection replay the core's status through the existing WebSocket path. Selection clears the previous Agent's display while the new snapshot is pending; frames for another Agent are ignored. A disconnected browser says it is disconnected instead of presenting an old snapshot as current. There is no browser-owned wake timer for reconnect to restart.

The source's accessibility decision follows Bill's reliance on speech and difficulty inspecting small visual distinctions. Text labels carry meaning that a colored dot would not, with file, delay, due time and error also available in the title and accessible label. The feature adds no automatic speech. Conversation and Actions remain available for the separate questions of why the Agent acted and what actually ran.

What the local evidence establishes

The implementation record describes tests using real Agents, skill loading, tool dispatch, rendering and owned files against local synthetic HTTP responses. They exercised same-turn goal edits, instruction placement, ordinary-input priority, stale callbacks, human stops, file repair, sandbox refusal and independent

background work. Short delays tested transitions; the eight-minute default was asserted separately, not observed through an eight-minute wait.

An isolated headless-browser run used the real GUI with synthetic model responses. It observed reconnect preserving the core due time, child selection showing no inherited goal, Interrupt turning reminders off, foreign-frame filtering and disconnected state after the owned server stopped. This establishes browser text, accessible attributes and measured layout, not human visual or hearing assessment; speech handling was transcript-only.

The source reports passing focused core race tests, the full GUI race suite and affected static and compilation checks after the corrections. These are bounded local results, without a live provider, real credentials or a running user session. They establish neither deployment nor an installed or enabled goal in this collaboration. Synthetic responses can expose a duplicate instruction or a stale wake; they cannot establish that a real model will correctly decide whether to continue, wait, stop or ask for permission.

Epilogue: What the student changed

The first attempt to revise the book failed before the student could provide the intended lesson. An author was placed ahead of the student to improve each assignment. A small persistence exercise became a larger system: 254 physical lines of production Go in the original implementation, 4,585 in its replacement, excluding tests and measured against each version's predecessor. The reviewer approved the expansion. Bill stopped the attempt and preserved it for inspection.

The original epilogue records the problem plainly: the review had checked whether the student built the revised assignment correctly, not whether the revision still served the original task. The failure wasn't that large programs are necessarily bad. The assignment had changed before its supposed improvement was tested.

The restart reversed that order. The student used the original book and graders, with access to its own preceding implementation but not the old answers. A reviewer compared implementations after the student had built and tried each chapter. Only then did the teaching change. That preserved a stable instruction against which a misunderstanding could be explained.

Corrections with a cause

The tool-loop assignment told the student to record text and tool calls, then continue until the model stopped requesting tools. The student also concatenated intermediate narration into the answer returned to the caller. The unchanged grader rejected it. Returning only the final response's text raised the recorded score from 55 to 90 out of 100; a separate inherited media check still failed. The corrected teaching distinguishes what belongs in conversation history from what the caller receives.

The memory assignment contained a different error. It described a compression trigger using cumulative input tokens. A lifetime total cannot fall when context is compressed. Bill clarified that the trigger should use input from the preceding request, including cached input once. The authors checked the accounting contracts and original implementation, recorded the assistance, and corrected the explanation. Here the student's difficulty exposed a wrong instruction, not a wrong implementation.

A Gateway check exposed a wrong test. A deliberately defective implementation returned the expected refusal to a child's key-registration request while still registering the forbidden key. The revised grader initially passed it. Trying that key afterward revealed the unauthorized access. The stronger check rejected the same defect, and the initial passing result remained in the record to explain why the test had changed.

These are three different repairs: clarify the result contract, correct the measurement, observe the forbidden consequence. Treating all three as “the model made a mistake” would discard the knowledge needed to prevent their recurrence.

Beyond the construction tests

Daily use added another kind of evidence. As Chapter 25 records, the second-edition crossover exposed an old executable behind a changed speech control, stale JavaScript after a server restart, and a total HTTP deadline that killed legitimate responses. A source edit, a build, an installation and an observed running change were separate events. Construction tests hadn't eliminated those integration failures.

That crossover advances beyond the original epilogue's then-unfinished plan. It doesn't establish a recurring autonomous improvement cycle, or new Edition 3 execution results. Likewise, the local tests behind Goals establish timer, admission and display behavior, not that a real model will reliably choose the right next action. A reminder provides another opportunity to inspect unfinished work; it doesn't supply the missing judgment.

The durable result is more specific than a successful demonstration. The next implementation can distinguish an answer from its history, current context from lifetime usage, a refusal message from denied access, and saved instructions from evidence of completed work. The codebook makes those distinctions executable. This companion preserves the failures and reasons behind them, so the next technical lead can change the implementation without having to rediscover why they matter.