

IDYLLIC LABS

Elements of Agentic System Design

William Chen



SECOND EDITION

WORKING DRAFT

Contents

William Chen · Idyllic Labs

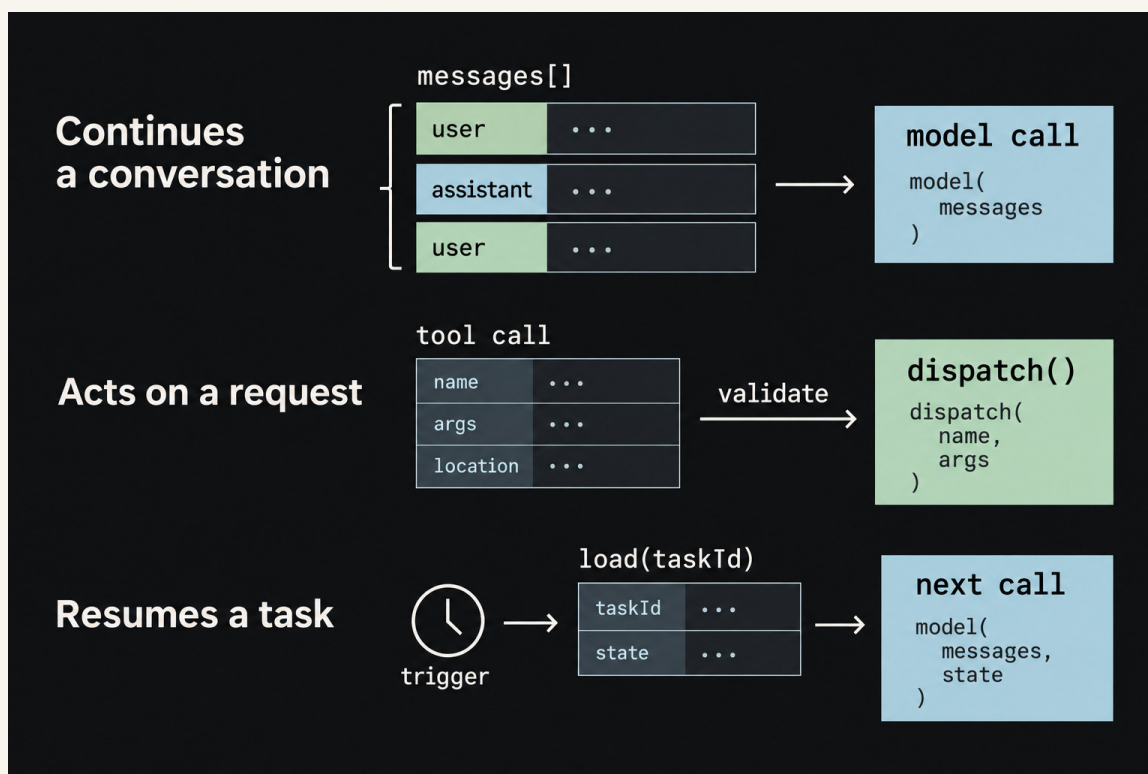
Introduction	3
1. Context	7
2. Memory	16
3. Agency	29
4. Reasoning	37
5. Coordination	47
6. Artifacts	57
7. Autonomy	69
8. Evaluation	78
9. Feedback	90
10. Learning	102
Further reading	113
About this edition	114

Introduction

This book introduces the design of agentic systems: programs that can choose and carry out actions in pursuit of a goal. We focus on systems built with large language models (LLMs), which generate responses to supplied instructions and information. The surrounding program can use a model's response to choose an action and carry out that action. The design of the program determines what information the model receives and what actions the system can perform.

You have considerable freedom in how you build such a program. You can ask a model to draft an answer, then give the draft and its sources to the model for review. You can also divide a large assignment into smaller tasks and request a separate response for each task. When does splitting the work help? We choose an arrangement by identifying the problem each component solves and weighing the cost of connecting the components.

Elements of Agentic System Design begins with a single model call: one request to a model and the response the model generates. We connect calls into agents that can take actions and use the results to decide what to do next. The book then develops larger organizations of agents, including systems that operate autonomously and learn from the outcomes of their work. Each new structure addresses a limitation or requirement that the simpler design brings into view. The elements organize this progression around engineering problems that recur as systems grow.



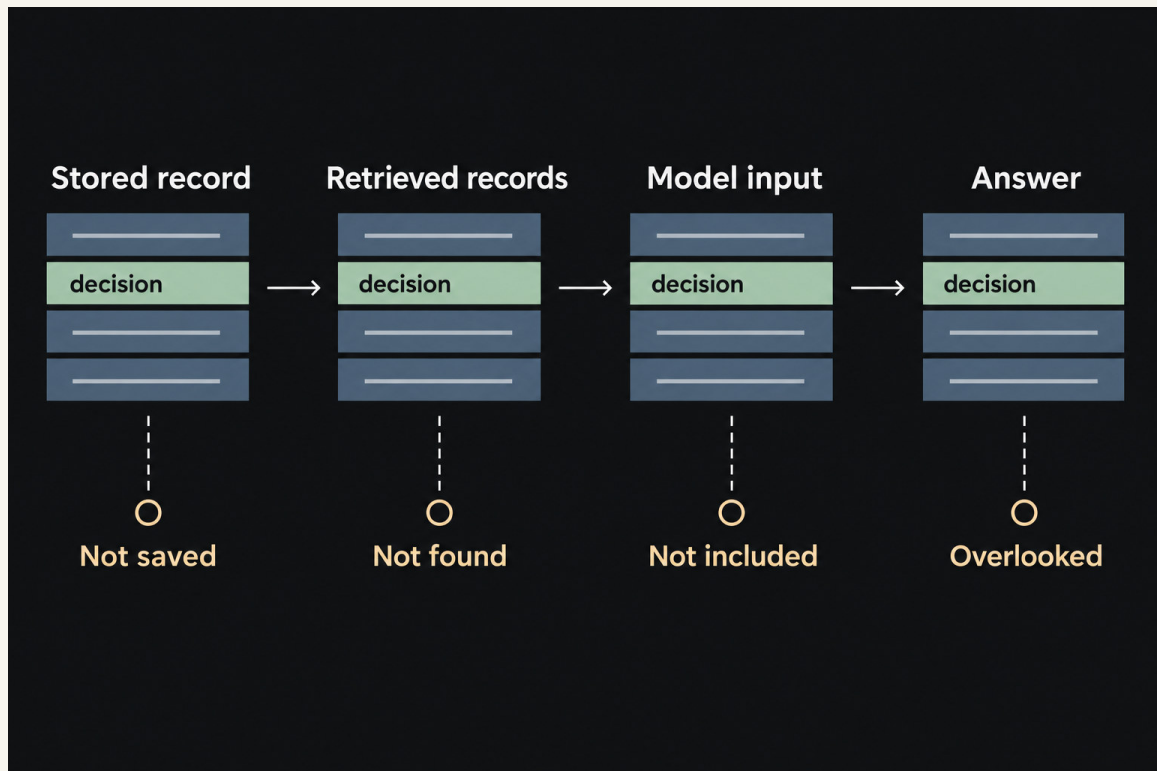
A conversation, an executed action, and a resumed task each add a connection around the same basic unit: a model call.

Context offers an early example of how a limitation shapes a design. A model's context is the information available during a call. Because that context has a size limit, a large task may benefit from several calls, each with the instructions and evidence for one part of the work.

The same constraint can motivate separate agents for research, drafting, and review. Each agent needs the assignment's requirements and relevant work from the others. Dividing the work therefore creates another design problem: how to distribute information and assemble the context each agent needs.

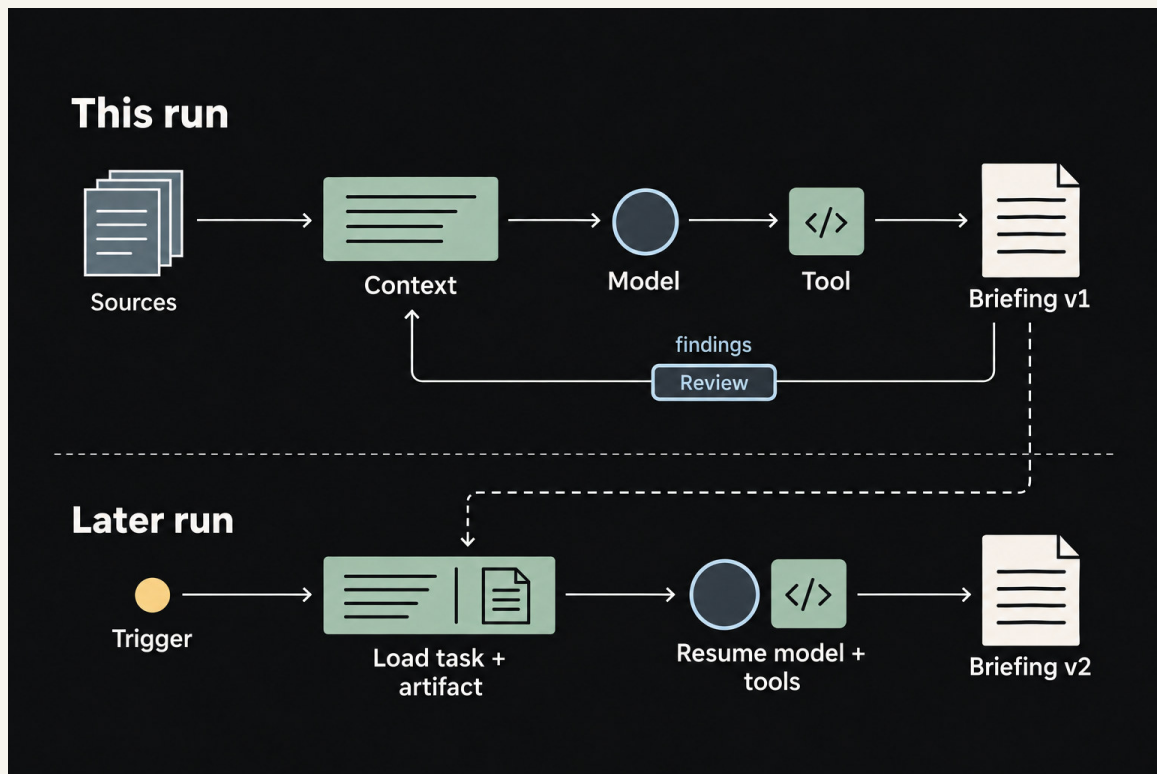
The elements also connect familiar cognitive behaviors to simple code structures around model calls. Memory requires stored records and code that selects the information needed for a later call. An agent's cycle of decisions and actions requires a loop that returns each action's result to the model. These connections make failures easier to investigate. If an agent ignores an earlier decision, we can check whether the program saved the decision and included it in the next call. A

missing record requires a different fix from a model that received the decision but failed to follow it.



Information reaches a call through several operations. A record surviving in storage is only the first step; selection and input construction determine whether it can be used.

The book assumes curiosity about software, but no experience building agents. Diagrams show how the components connect and how information moves between them. Examples of a research assistant gathering sources and writing a briefing show how the elements work together.



One example combines the structures developed in this book: calls and tools within a run, a review loop, and saved work that another run can resume.

By the end of the book, you should be able to explain why an agentic system needs each of its components and how a different arrangement would change its behavior. That understanding gives you a basis for adapting existing patterns and inventing new ones as models improve. The first chapter, *Context*, examines how the information supplied to one model call shapes its response.

Context

What information is available at a particular step.



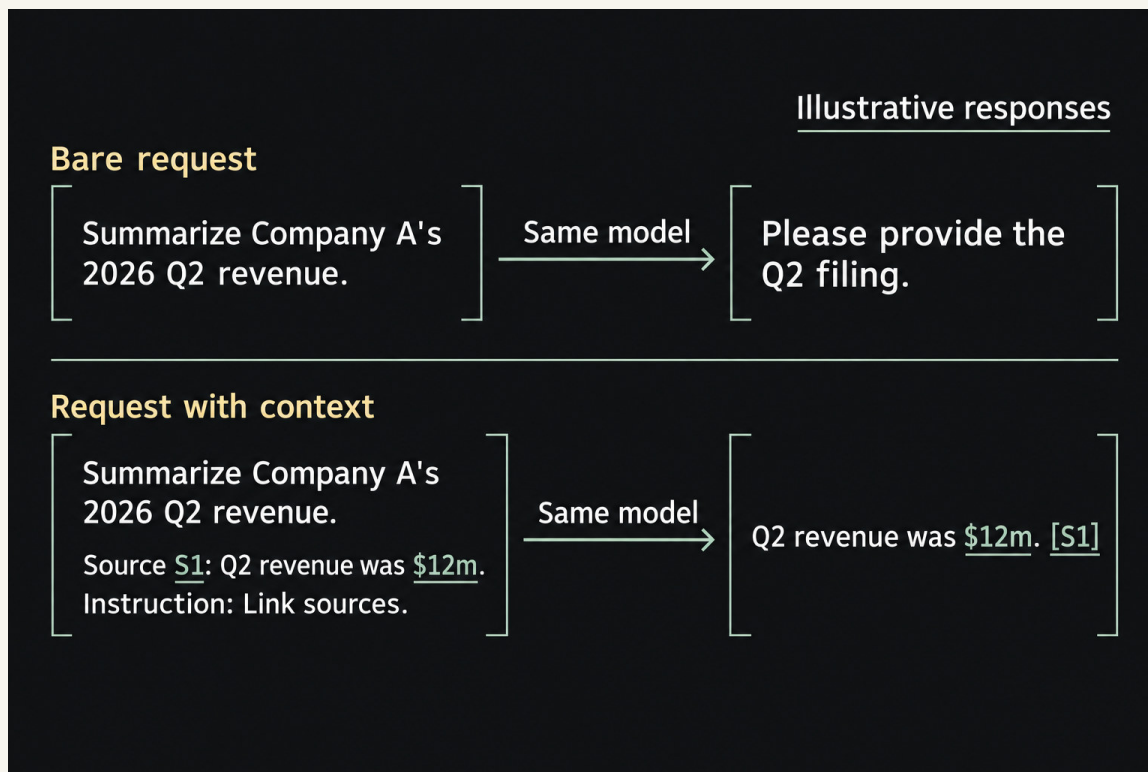
Each model call receives a selected part of the information available to the system.

Context is the input available while the model computes a response. It includes the current request, instructions for how to respond, and any material supplied with them: earlier messages, documents, tool results, or supported images and audio. The model uses this input together with capabilities and knowledge acquired during training to produce its next response.

One invocation is one use of the model to produce a response. For a given model, changing its input is the main way an application influences that response. The same question can produce a different answer when accompanied by a source document, a worked example, or an instruction to explain it for a beginner. Context supplies both the information the model can work from and the description of what it is being asked to do. Before arranging several calls into an agent, we need to understand what one call can do with the input we give it.

A model asked to write a research briefing from a bare request has only its trained knowledge to draw on. Additional context gives it more to work with:

- **Company filings** supply source material for factual claims.
- A **citation preference** describes how to cite those sources.
- **Earlier messages** let the model continue a conversation about the draft.



Illustrative responses to the same request show how a supplied filing and citation instruction can change what a model returns.

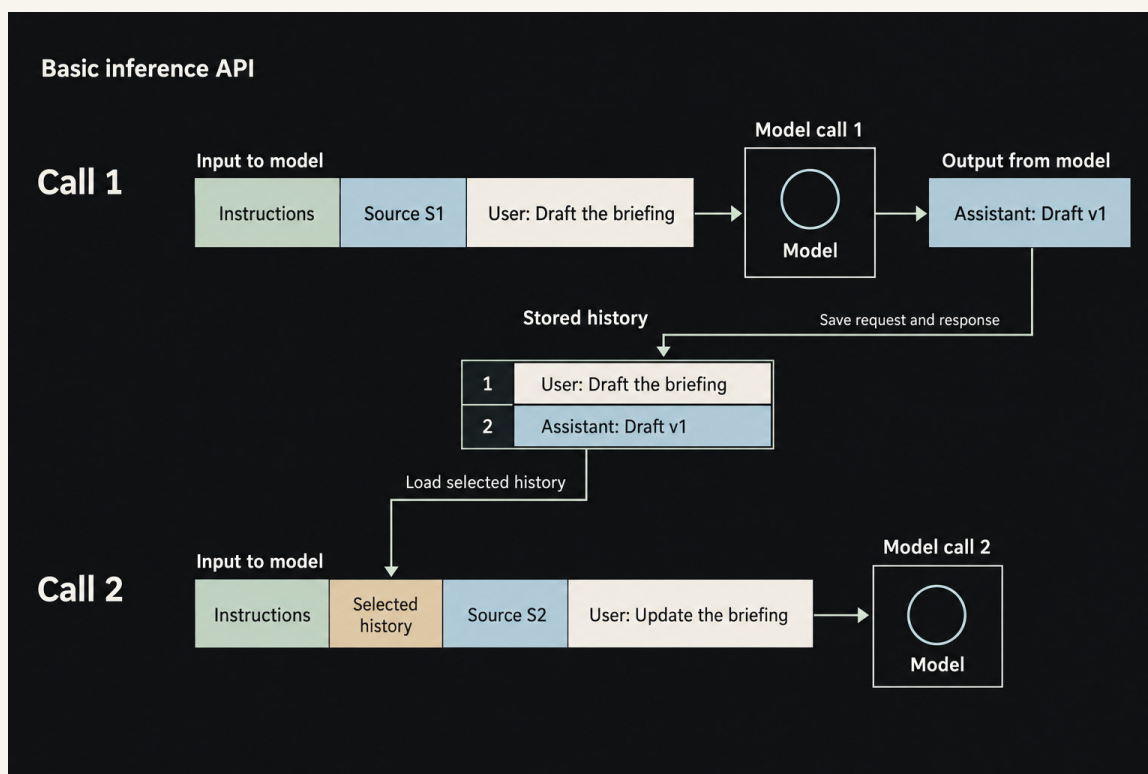
Conversation continuity

This raises a question about the next response. Once the assistant has read the filings and written a draft, how does it know what happened when the user returns? In a basic inference call, the model does not retain a personal task history for the next invocation. The application has to supply the relevant past again. The apparent continuity of the conversation comes from the model interpreting that supplied history as the situation it is continuing.

The simplest implementation stores a conversation as an array of messages. After each response, the application appends the user's request and the assistant's answer. For the next call it sends those messages along with the new request and the instructions that should continue to apply. In this illustrative input record, the assistant's knowledge of the previous exchange is visible in history; its citation preference and source material enter through separate fields.

```
// An application record; a provider adapter formats the actual request.
const input = {
  instructions: "Write a source-backed briefing. Cite factual claims.",
  profile: { citationStyle: "linked sources" },
  evidence: sources.map(({ id, text }) => ({ id, text })),
  messages: [
    ...history,
    { role: "user", content: "Update the briefing with this filing." },
  ],
}
```

A provider can manage this history behind a conversation identifier, and caching can avoid recomputing an unchanged prefix. Those services change who stores the history and how it is processed. They do not remove the need to make the relevant past available to the next invocation. A later call might receive the full transcript, selected turns, or a summary. What the assistant can continue from depends on what that reconstruction preserves.



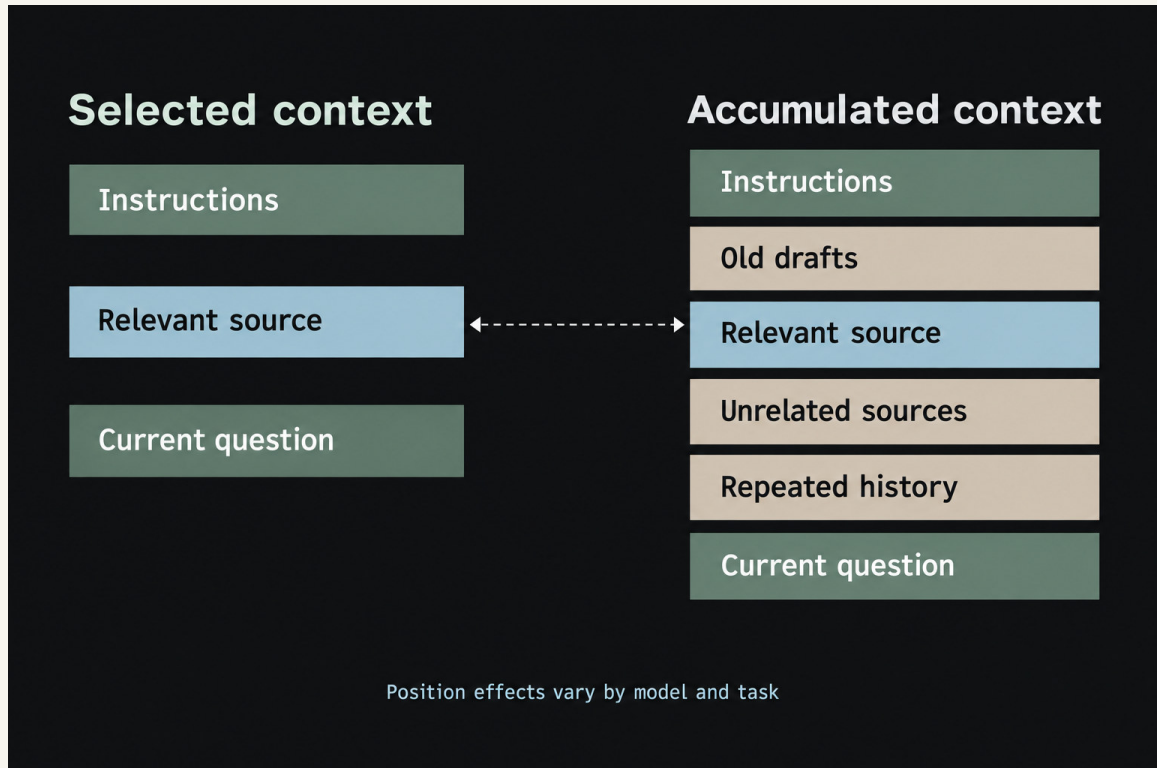
A basic inference API receives the history the application supplies. Persisted messages connect separate calls.

Context limits

Resending the history works while the conversation is short. Over time, the transcript accumulates requests, drafts, corrections, and tool results; the source collection grows alongside it. A model's context window has a finite capacity, measured in tokens. Tokens are the units in which text and other supported content are represented for the model. An application must fit its input within the model's limits and leave room for generation where the limit is shared.

The briefing may eventually depend on more material than any one call can contain. The application then has to choose: which earlier decision matters now, which source passage supports this claim, and which old draft can be left out? Taking the most recent messages is easy to implement, but a decision made last week may matter more than today's failed searches. The application must select the parts of the larger record that this call needs.

A larger window postpones the capacity limit, but does not settle that selection problem. Additional input can increase processing time and cost; irrelevant or conflicting material can make useful evidence harder to use. Research on long contexts has also found that the position of evidence can affect performance, with the effect varying by model and task. A passage fitting in the window establishes that it is available, not that the model will notice or interpret it correctly.



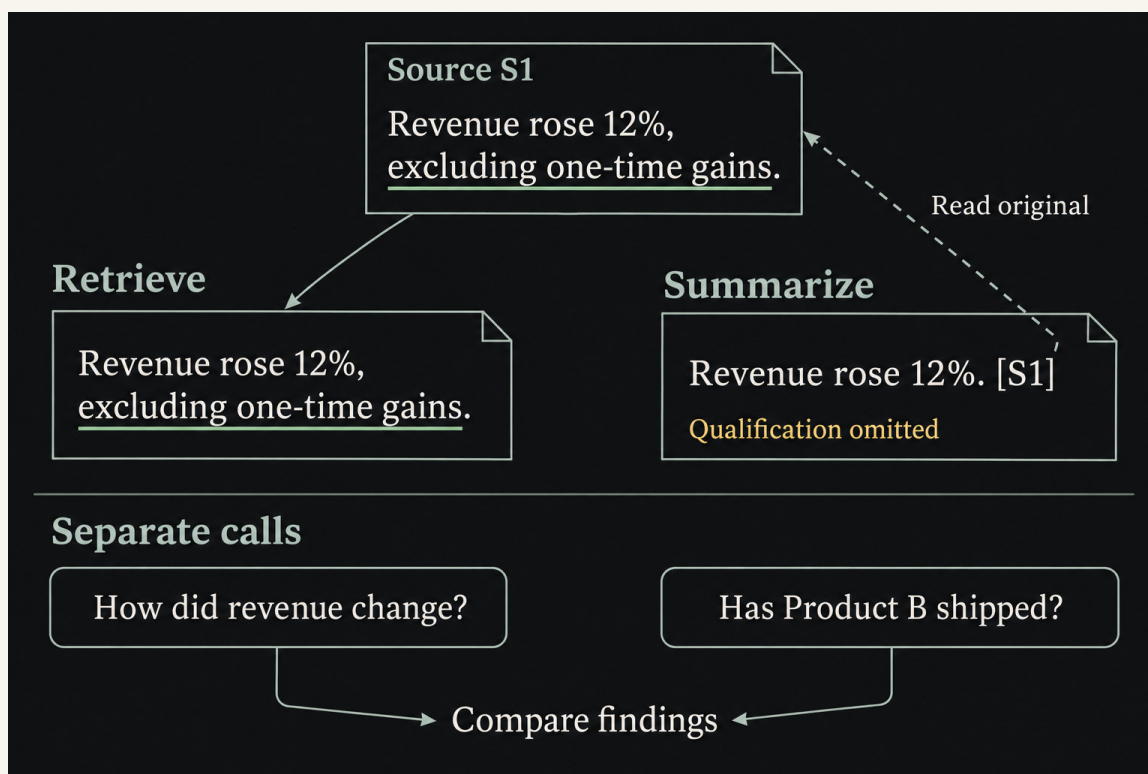
A larger input can bury the same useful evidence among unrelated material. Context selection must be tested against the task.

Context management

One response to this limit is retrieval: keep the full collection elsewhere and fetch the passages needed for the current question. Another is compression: replace a long exchange with a summary of its decisions. These operations preserve different things. Retrieval can retain the exact wording of a selected passage; a summary preserves an interpretation while discarding detail. Keeping source references lets a later call return to the original when the shortened account is insufficient. Instructions can be selected too: loading a reusable procedure, often called a skill, when needed avoids carrying every procedure in every call.

Sometimes even the relevant material is too much for one useful input. The work can then be divided into questions with separate contexts. One call examines financial filings, another examines product documentation, and a later call compares their findings. Each receives a narrower job and the evidence needed for it. This creates room to investigate more material, but now the findings must be passed between calls without losing the qualifications the comparison needs.

Separate calls can be stages of one process; each call need not be a separate agent. An agent maintains a task process that can make further calls and choose operations as work proceeds. When several such processes need different evidence or responsibilities, giving them separate contexts can help. That division also introduces handoffs and coordination costs. Later chapters explain how calls become agent loops and when a task benefits from more than one.



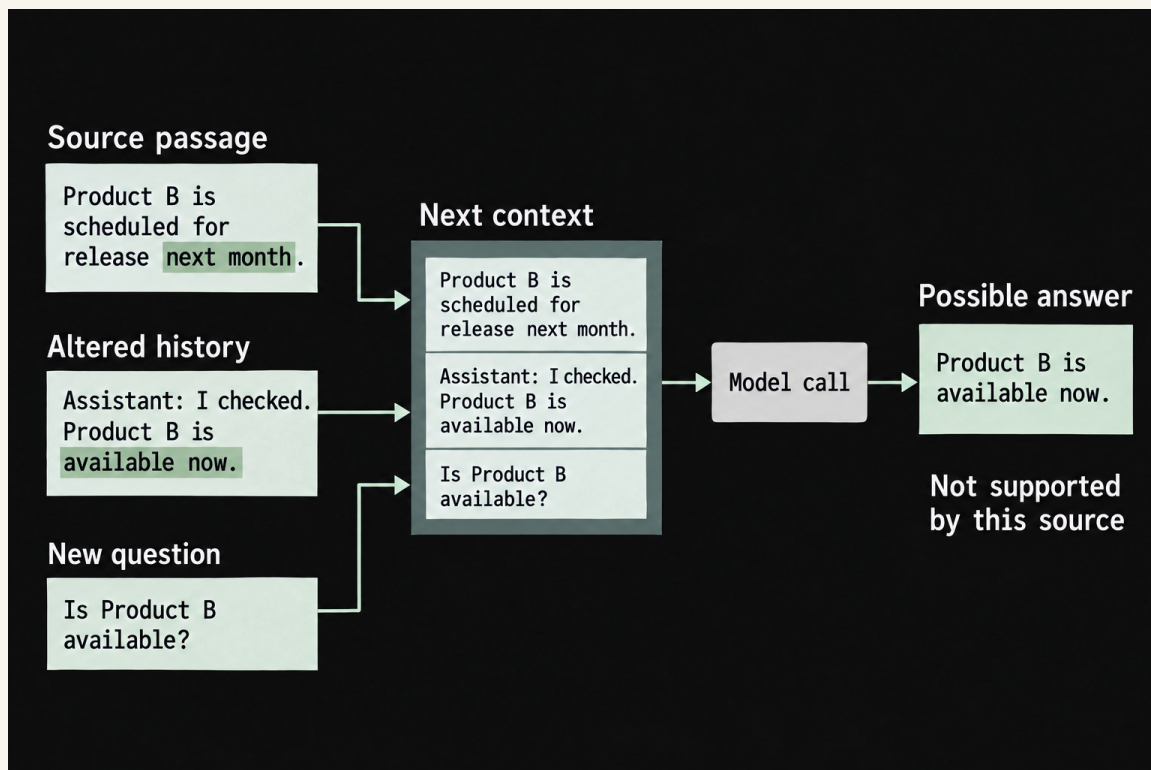
Retrieval can preserve a passage verbatim. A summary can omit a qualification while retaining a reference, and separate calls can investigate different subquestions.

Grounding and prompt injection

Selection affects factual reliability as well as capacity. The model may know how financial statements work without knowing the figure in a newly released filing. If that filing is absent, it lacks the evidence needed to answer a question about the figure. It may report uncertainty, or it may produce a plausible unsupported answer. Supplying a relevant source gives the response a factual basis, a practice called grounding. Grounding reduces this particular source of error; it does not prevent misreading, faulty inference, or contradictions between sources.

The same sensitivity to supplied information creates an attack surface. A retrieved document can contain instructions aimed at the assistant. An altered conversation record can falsely show that the user supplied a figure or that the assistant already checked a source. Because reconstructed history helps establish what the model thinks is happening, changing that history can change its next decision. The place where the application builds context is therefore also a place where source identity and trust matter.

The application can preserve who supplied each message and label retrieved passages as source material. Message roles and instruction hierarchy help distinguish task instructions from text being examined, but the model can still misinterpret a source's instructions as directions. Keeping original message records and source references makes it possible to inspect what the next call actually received. Context construction determines both the evidence available and the situation the model is asked to continue.



The source describes a future release, while an altered history claims current availability. Supplying both exposes a conflict; the next call can still repeat the unsupported claim.

Context records

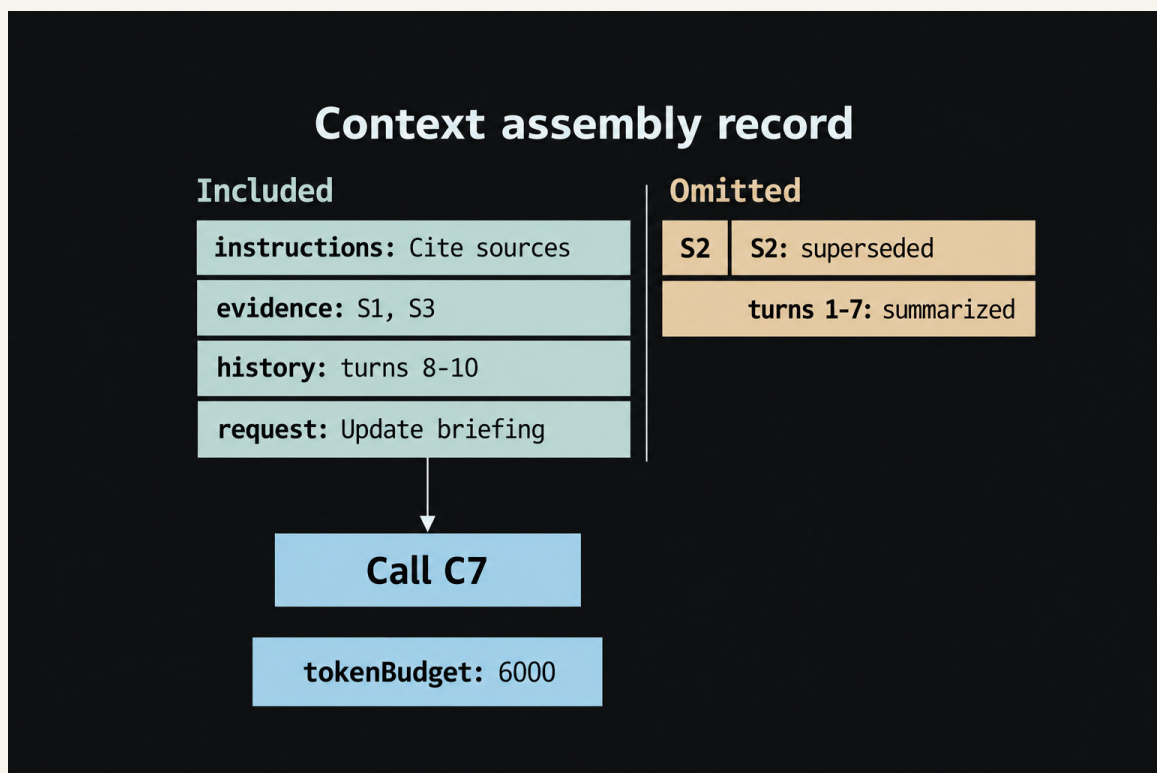
A context builder can make these choices before calling the model. Its job is to combine instructions, the current request, relevant history, and selected evidence within a budget. Recording what it selected and omitted makes the process inspectable. The record below describes those decisions; an adapter still has to turn them into the provider's supported message and content format.

```

type CallContext = {
  instructions: string[]
  evidence: { id: string; text: string }[]
  history: { role: "user" | "assistant"; content: string }[]
  request: string
  inputTokenBudget: number
  omitted: { id: string; reason: string }[]
}

```

If the briefing omits an earlier decision, this record helps distinguish a missing input from a failure to use an input that was present. The next question is where that decision was stored and how the builder looked for it. A finite context is a temporary view of the work; preserving the larger record and finding the right part of it is the problem of memory.



The context builder records what reaches a call and what it leaves out. The input budget shown here is illustrative; selected history and source references make the construction inspectable.

Memory

How information survives and becomes available again.

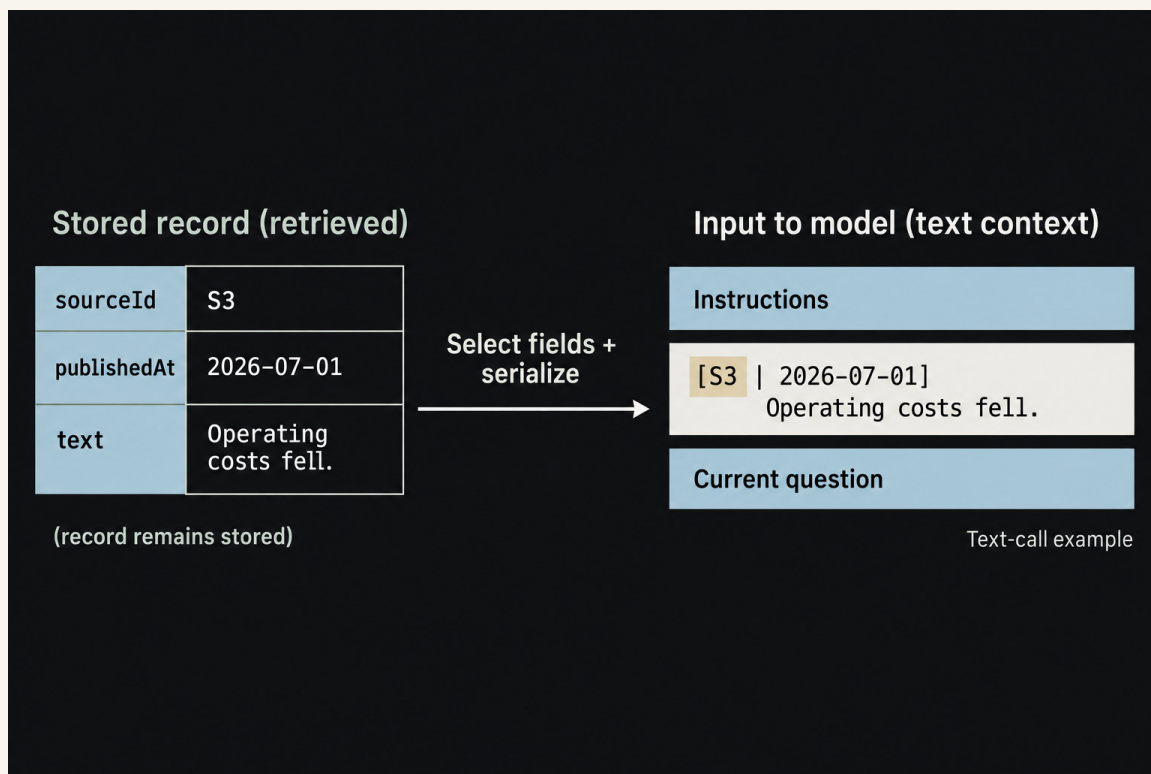


Stored records outlast a call. Retrieval brings selected records back into context.

Memory preserves information beyond the context of a single model call. A conversation, project, or collection of sources can grow larger than one context window. The system therefore needs somewhere to keep that information and a way to bring the useful parts back into context. What the model appears to remember depends on what the system saves, what it retrieves, and how the next call interprets it.

Three design questions

1. **What should be stored?** The record needs to become useful input for a later model call. In the text examples here, that means retaining text or data that can be presented as text. Notes, observations, and explanations of failed attempts can all be useful if they preserve enough information for the model to interpret them in a new situation.
2. **How should it be stored?** How the information will be retrieved helps determine its format. A Markdown file suits reading a profile or procedure; database rows make filtering events by user and time straightforward. The storage format need not match the input format: selected fields can become a passage in context. The useful choice is the one that makes the needed information easy to recover.
3. **How should the model find and access it?** Folder names, short descriptions, and searchable indexes help the model recognize relevant material before reading it. The application can supply a small catalog or frequently needed facts at the start, then retrieve other material when the model requests it. Preloading saves discovery steps but consumes context; access on demand saves space but requires a way to find what is missing.



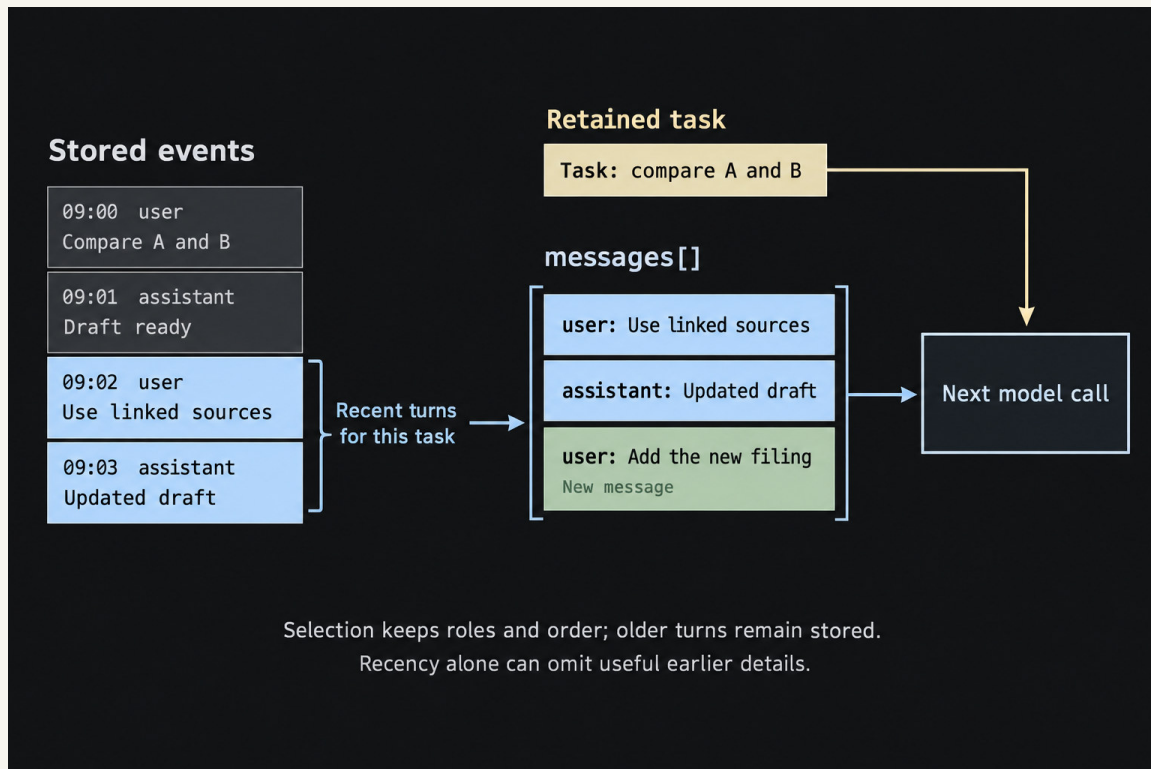
A retrieved record becomes text in the next context. The model interprets the passage alongside its source, date, and the current question.

Conversation history

Conversation history is the most familiar example. An application saves each message with its role, such as user or assistant. When a new request arrives, it supplies the earlier messages along with that request. The model reads the exchange and responds as its continuation. For a short conversation, the saved history can be an array that the application persists between requests.

```
const history = await conversations.load(taskId)
const messages = [
  ...history,
  { role: "user", content: "Add the new filing." },
]
const response = await callModel({ messages })
await conversations.save(taskId, [
  ...messages,
  { role: "assistant", content: response.text },
])
```

Longer histories can live in an event log, where each entry records a message or tool result with its task and time. Code can query recent turns instead of loading the entire log. This is already a retrieval policy: recency decides what the next call sees. Selection must preserve usable exchanges, including the results associated with tool calls. Earlier events remain in storage even when they no longer fit in the current input.



Stored events preserve the exchange. Selected turns keep their roles and order, while a retained task record carries the continuing assignment.

Decision records

Recent history eventually leaves important decisions out. A separate note can retain a decision and its reason after the conversation moves on. “Used the cached filing” records an action. “The live endpoint returned a rate-limit error; used the cached July filing” also preserves why the workaround was chosen. A later call can interpret that reason and judge whether to try the live endpoint again. The note needs enough evidence to support that judgment, even if its wording is rough.

User profiles

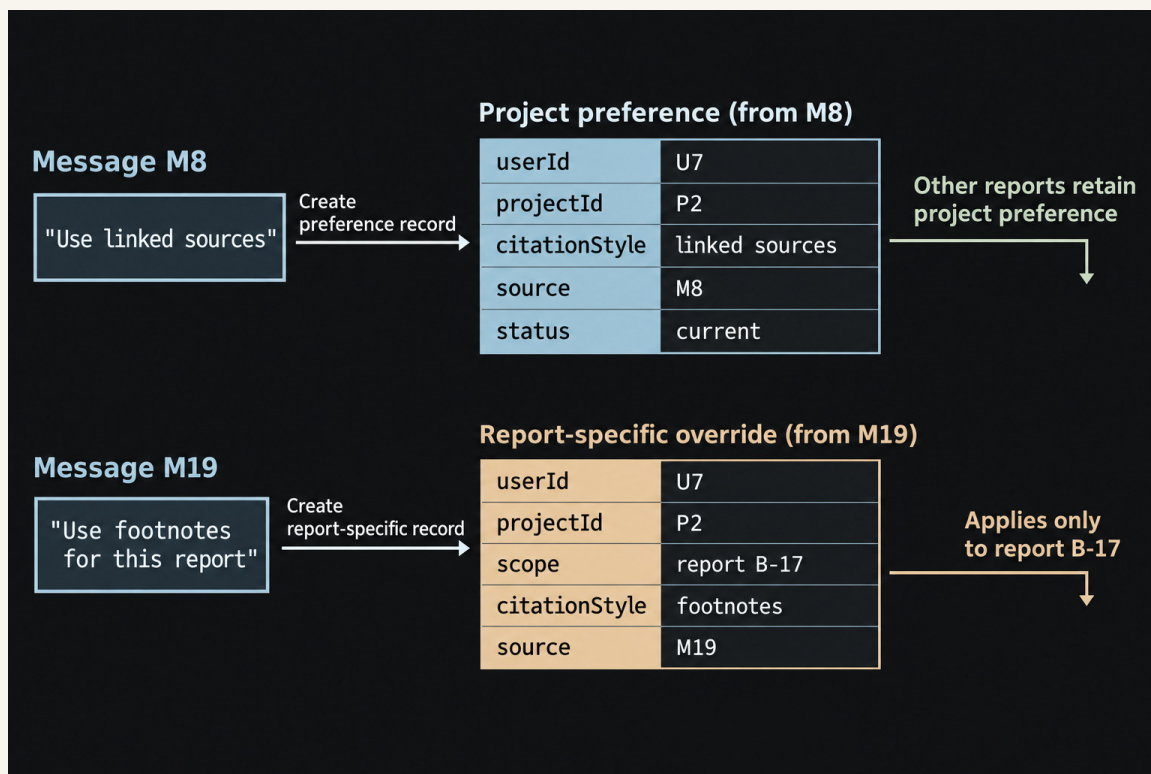
A user profile can begin as a Markdown note. If a user asks for linked sources throughout a project, the application can save that preference in `users/U7/profile.md` and read it when preparing later requests. The note can preserve the user's wording, the project it applies to, and the message that established it. A model can interpret those details without a separate database field for every possible preference.

```
# User U7

## Project P2
Use linked sources in research briefings.
Requested in message M8 on 2026-07-01.

## Report B-17
Use footnotes for this report.
Requested in message M19; other reports keep linked sources.
```

The note records where each preference applies and where it came from. The report-specific request overrides the project preference for one report; it does not establish a new preference for everything. Retaining the source message lets the application check or correct the note. As the profile grows, it can split into sections or files so a request about one project does not load every fact about the user.



A stored preference should preserve who supplied it, where it applies, and the evidence for changing it.

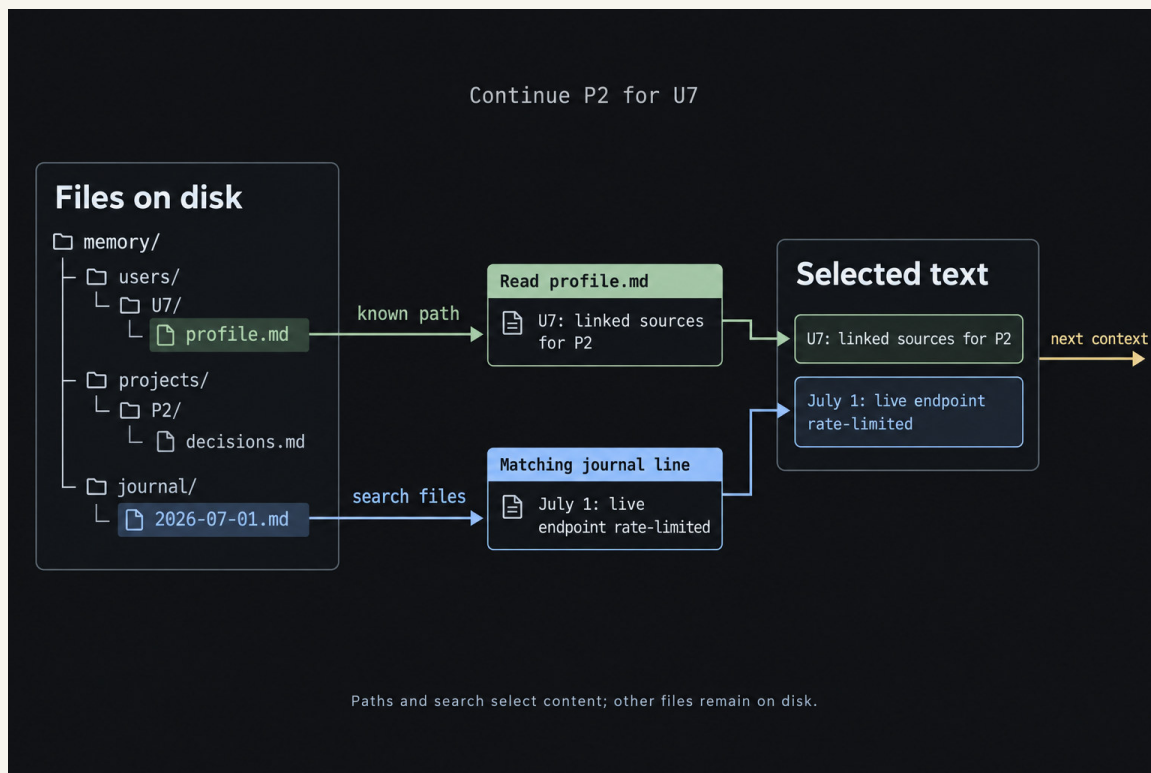
Folders and text search

Meaningful paths provide a simple index. A user identifier locates a profile; a project directory collects decisions and open questions; a dated filename locates a journal entry. A directory listing tells the model what records exist without supplying their full contents. The application can read a known path directly, or let the model choose a file to inspect through a reading tool.

```
memory/
  users/U7/profile.md
  projects/P2/decisions.md
  projects/P2/open-questions.md
  journal/2026-07-01.md
  journal/2026-07-02.md
```

When the path is unknown, a text search such as `grep` or `rg` can find matching lines across the files. Paths and line numbers let the model request the surrounding passage when a match is too short to interpret. This gives a small

collection a practical retrieval process: inspect names, search content, then read the useful part. A filesystem can support selective retrieval before a specialized search index is needed.



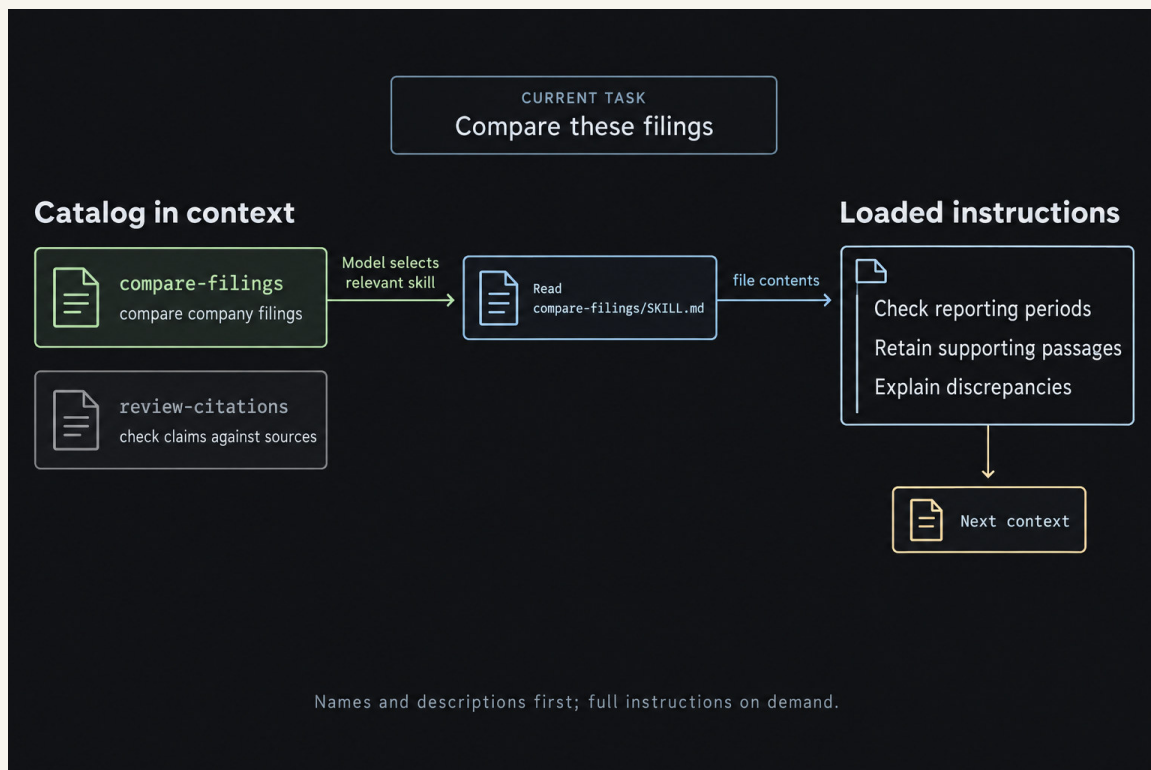
A known path locates a profile; text search locates a journal passage. Only the selected text enters the next context.

Skills for procedural memory

Some saved knowledge describes how to work. A skill can store a procedure in a `SKILL.md` file, with supporting scripts or examples nearby. A system with many skills cannot afford to include every full procedure in every context. It can instead supply a small catalog of names, descriptions, and locations. The model uses that metadata to judge which procedure is relevant to the current task.


```
const skillCatalog = [
  {
    name: "compare-filings",
    description: "Compare company filings across reporting periods",
    path: "skills/compare-filings/SKILL.md",
  },
  {
    name: "review-citations",
    description: "Check whether source passages support a draft's claims",
    path: "skills/review-citations/SKILL.md",
  },
]
```

For a filing comparison, the model can request the first file. Code reads it, and the next call receives the full instructions while the other skill remains only a catalog entry. The description matters because the model must recognize a useful procedure before reading it. This separates discovery from loading: a small index helps choose which larger body of knowledge should enter context. Agency will explain how a model's request becomes the file read; loading a procedure does not itself execute its steps.



The model sees skill names and descriptions, selects a relevant procedure, and receives its full instructions through a file read.

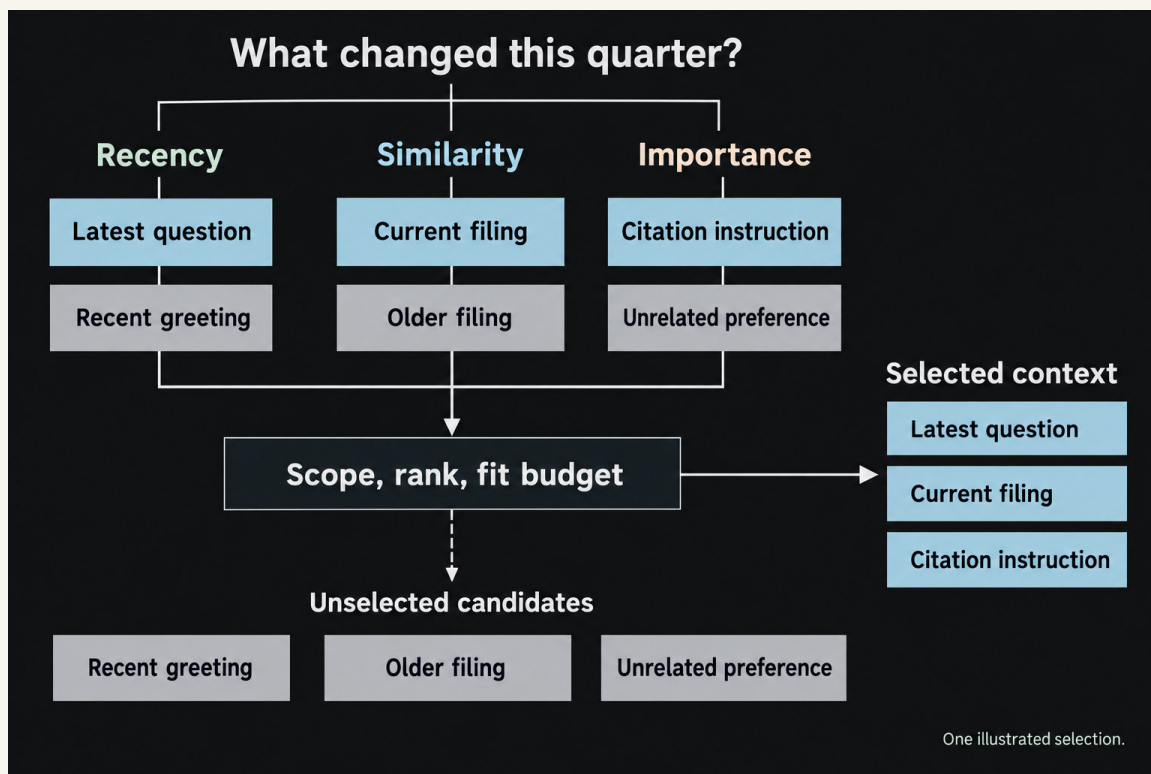
Journals

Profiles and procedures collect knowledge expected to remain useful. A journal instead preserves observations and decisions as work unfolds. Each entry can record its time, project, what happened, and links to evidence. Unlike the raw conversation log, it selects information worth keeping for future work. “What did we decide yesterday?” can use a date range; a question about a known endpoint can use text search.

Semantic search

Text search can miss an entry when the question and the entry use different words. “Why did the analysis get cheaper?” might need a note that says “Switched to discounted batch requests after the rate-limit errors.” Semantic search offers another retrieval route. An embedding model converts entries and queries into vectors that an index compares for similarity. The index returns candidate passages, so a related idea can be found even when the query shares few exact words with the note.

A search can find more candidates than the next context should contain. Reranking compares the candidates more closely with the current question, often using a model trained or prompted to assess relevance. Code can then choose a small set within the context budget. A recent entry, a semantically related older entry, and a standing preference can all matter for different reasons. The selected passages and their sources enter context for the model to interpret together. Retrieving material to support generation is commonly called retrieval-augmented generation, or RAG.



Similarity, recency, and task importance can propose different records. The selection policy determines which reach context.

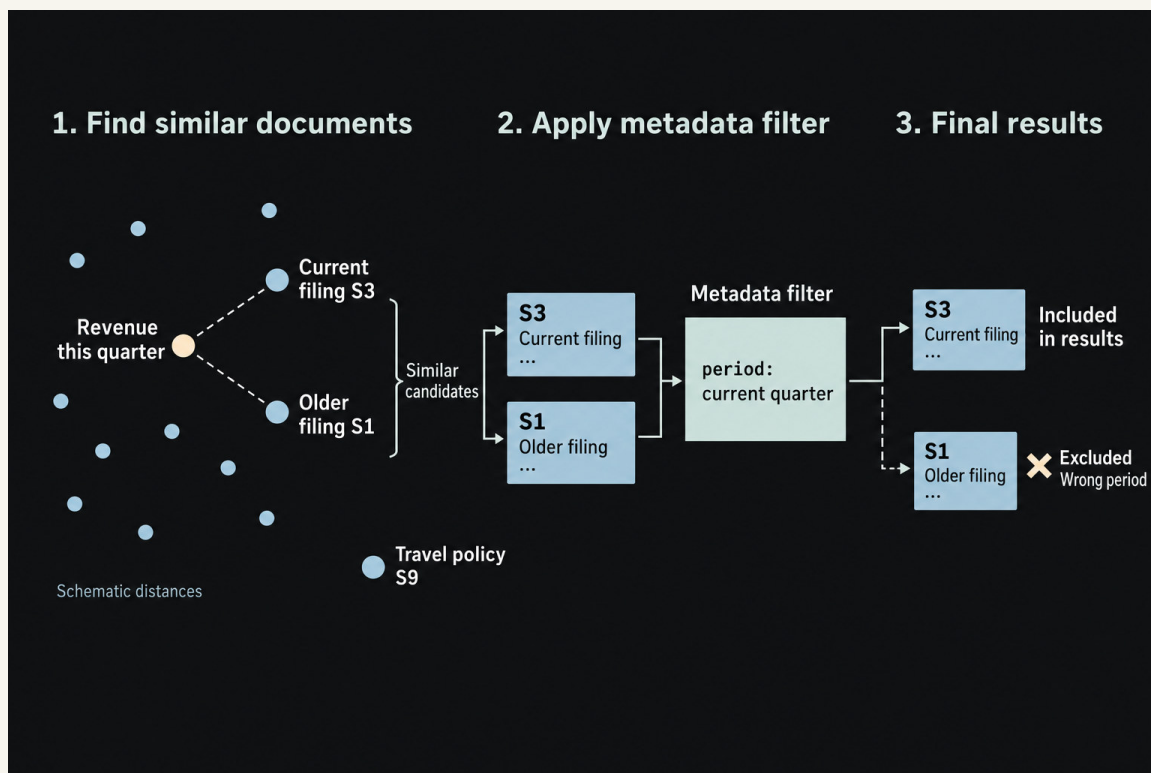
Retrieval quality

Relevance is a judgment about usefulness to a question, not proof that a passage is true or current. An old filing can closely match a question about this quarter's revenue. Filtering by company, reporting period, and access rights narrows the eligible material; retaining those fields beside the text lets the next call understand its scope. A successful search can still produce a wrong answer if those distinctions disappear when the passage enters context.

```
const evidenceText = selectedRecords.map(record => {
  const source = "Source: " + record.sourceId
  const period = "Reporting period: " + record.reportingPeriod
  const date = "Published: " + record.publishedAt
  return [source, period, date, record.text].join("\n")
}).join("\n\n")

// Text supplied to the next model call:
// Source: S3
// Reporting period: 2026 Q2
// Published: 2026-07-01
// Operating costs fell.
```

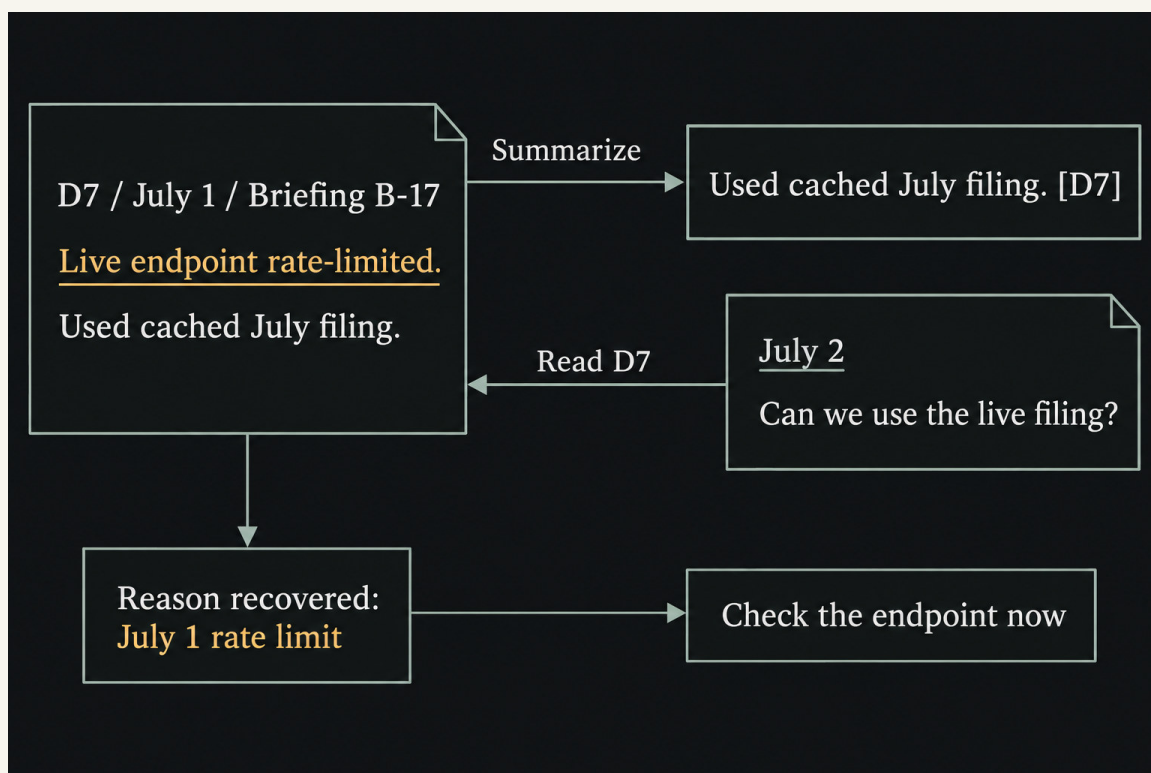
The model also has to reconcile what the retrieved passages say. A journal entry may record a temporary workaround, while a newer source shows that the original limitation has been removed. The model can use both to reconsider the earlier decision. It can also misread a date, miss a negation, or treat an inference as a fact.



A nearby embedding suggests a candidate. Date and project fields can still exclude it from the current task.

Different questions therefore benefit from different routes through the same stored material. A known profile needs a path lookup, a recent exchange needs a history query, and an unfamiliar description may need semantic search followed by reranking. The model can help choose the route and judge the results. The application still has to perform the reads and supply the selected text in a later call.

Summaries save space by retaining an interpretation of earlier material. Keeping references to the original lets a later call recover details that a new question makes relevant. The same principle applies to profiles and journals: preserve the reasons, dates, and scope that allow future interpretation, then provide retrieval paths that can find them. The storage choice follows what those future calls need to read.



A summary can retain a reference to the original decision note. Reading that note recovers why the decision was made; current conditions still require a fresh check.

We now have ways to preserve a conversation, retain knowledge, and retrieve useful material without loading it all at once. The model can identify a missing source or request a procedure, but its response does not perform the read. The

next chapter connects those decisions to executable operations so the model can choose what information to obtain and what actions to take.

Agency

How a proposed action becomes an effect in the world.



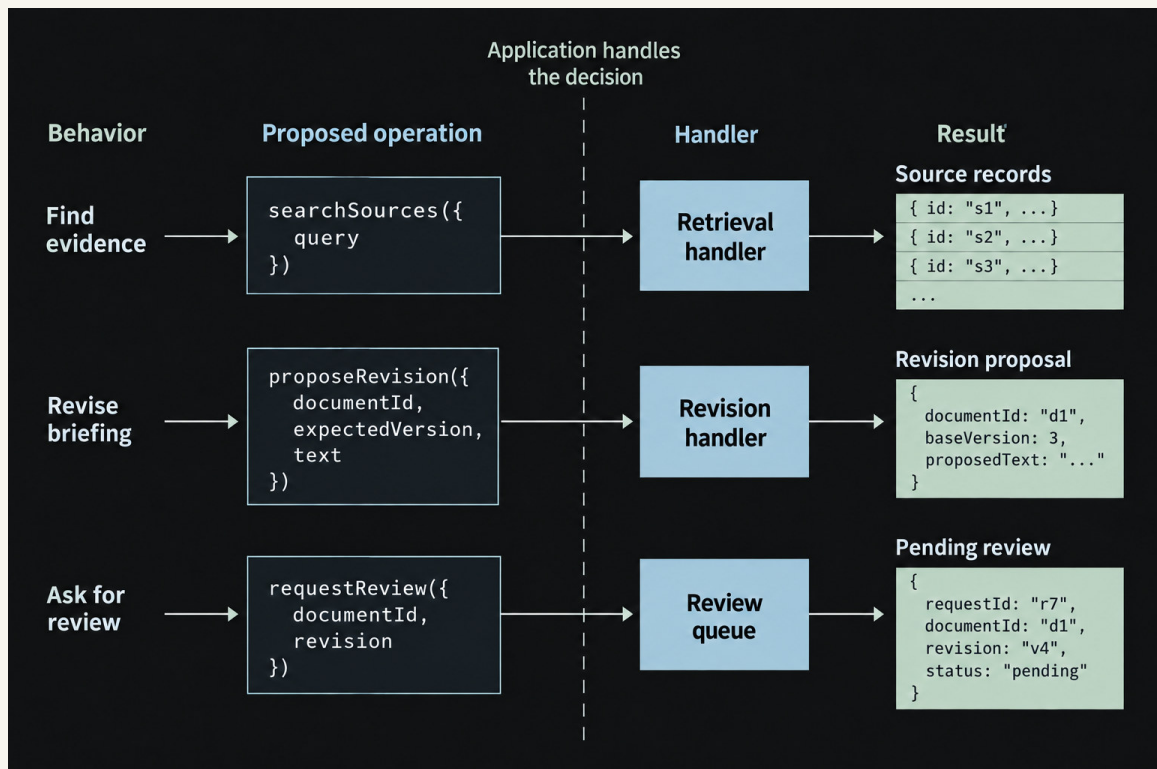
A model's decision has an effect when the surrounding program executes an operation.

Agency is the capacity to choose and carry out actions in an environment. Context and memory supply information to a model; agency connects the model's decisions to operations such as reading a file, querying a database, or editing a document. A model can generate a request for any of these operations, but application code must execute the request for the operation to happen.

The connection has three parts: a decision, its execution, and an observation of the result. The model chooses a search query, code runs the search, and the returned passages become input to a later call. The software around the model, often called a runtime or harness, connects these parts. It determines which requests the model can make, which requests actually run, and what the model learns about the outcome.

Three design questions

1. **What decisions can the model express?** The application supplies a vocabulary of operations and the information needed to choose among them. A named search function lets the model specify a query. A code-execution tool lets the model describe a computation as a program. The available forms determine how precisely and flexibly the model can request work.
2. **What executes those decisions?** A request needs an implementation: a function, an interpreter, or a service. The application must recognize the request, check its arguments, and route it to that implementation. The implementation and the resources it can reach determine what the operation can do.
3. **What observation returns to the model?** The next call needs evidence of what happened: source passages, a saved revision, or an error. A request to perform an action is not evidence that the action succeeded. The returned information must let the model decide whether to continue, revise the request, or stop.



The application connects each model request to a handler and returns the result.

Tool calling

Tool calling is the common way to make this connection. A tool definition gives the model an operation's name, a description of its purpose, and a schema specifying its arguments. For example, `searchSources` accepts a query string and returns source passages. The model can request that operation with a query such as “latest company filing.” A handler is the function that receives those arguments and runs the search.

A tool registry connects each definition to its handler and argument validator. The application supplies the definitions to the model, then uses the registry to dispatch a returned request. The example below uses application interfaces; a provider adapter translates between these values and the model provider's message format. The observation retains the original call, including its identifier, so a later response can associate the result with the right request.

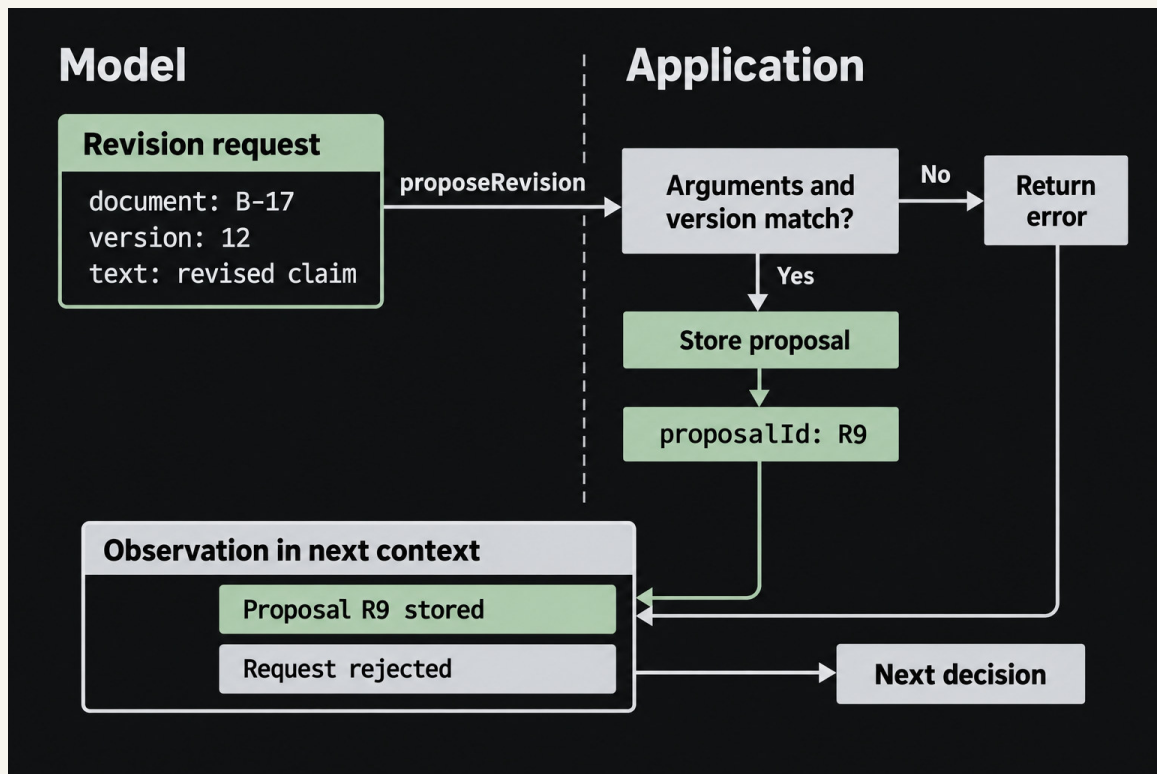
```
const decision = await callModel({
  messages,
  tools: registry.definitions(),
})

if (decision.kind === "tool") {
  const call = decision.toolCall
  const handler = registry.get(call.name)
  if (!handler) throw new Error("Unknown tool")

  const args = handler.parse(call.arguments)
  const result = await handler.run(args)

  observations.push({ call, result })
  // Include the request and its result when constructing the next call.
}
```

A tool's arguments determine how precisely the model can specify an action. An edit needs a document identifier and new text; an edit to a shared document may also need `expectedVersion`. If the model read revision 12 and another writer has since saved revision 13, the handler can reject the stale edit. That extra argument addresses a specific problem: the document can change between the model reading it and the application applying the model's decision.

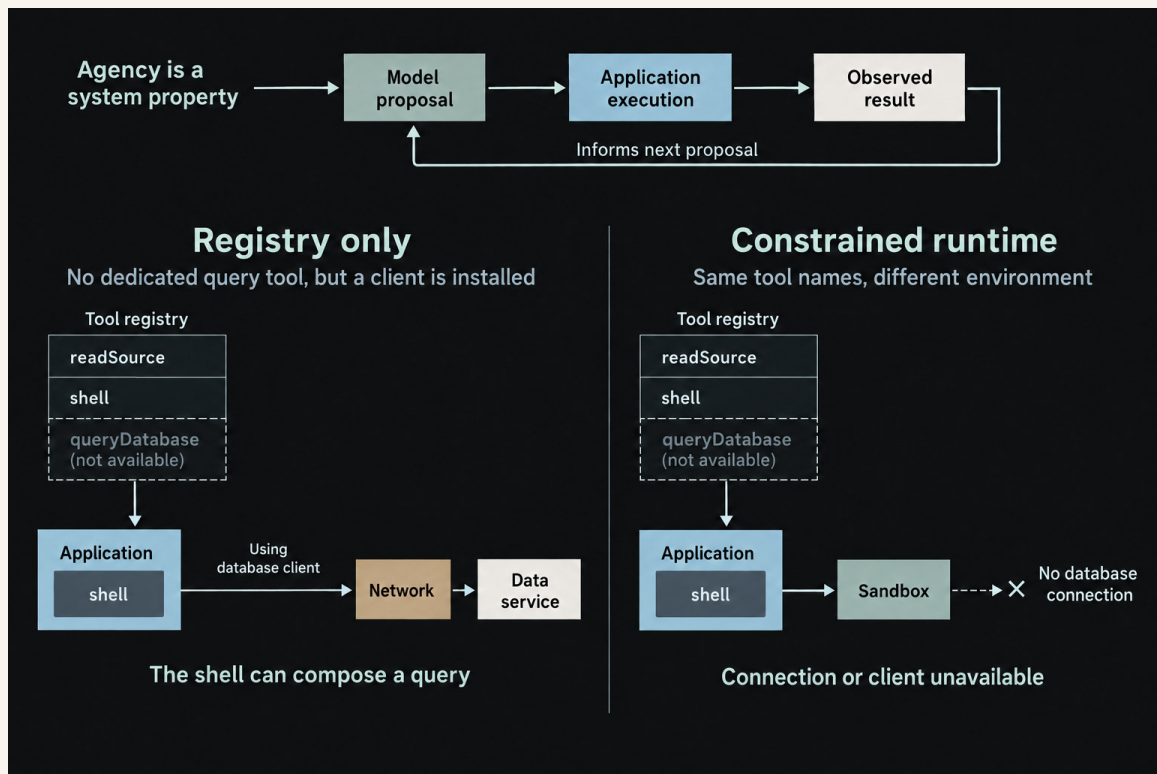


The handler checks the revision request before storing a proposal. The next context receives either the stored proposal identifier or an error rejecting the request.

Code execution

Named tools work well for operations with known inputs and outputs. Some tasks need combinations that would be cumbersome to describe as individual calls. To compare monthly totals in two CSV files, a model can write a short program that reads both files, groups their rows, and calculates the differences. An execution tool runs the program and returns the result. Loops, conditions, and transformations happen inside that execution, without a separate model call for each step. Tool calling still carries the request; the program is one of its arguments.

A shell extends the same idea to installed commands and pipelines: the model can search files, run a test suite, or invoke a command-line client. What those commands can do depends on the environment's files, libraries, network access, and credentials. A shell with a database client can query a data service even when the registry has no dedicated query tool. A sandbox can restrict those resources and operations.



A shell can use installed clients and available connections even when the registry has no dedicated tool for the operation.

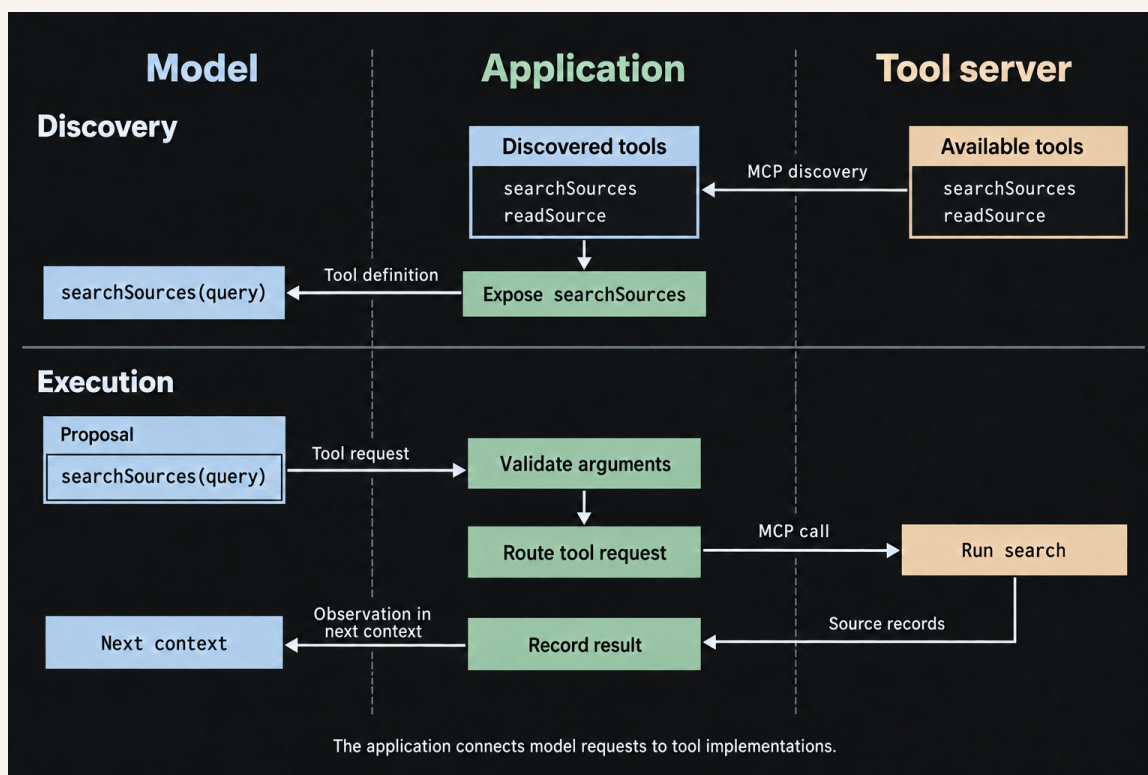
Loading skills

An action can also change what the next model call knows. In the Memory chapter, a skill catalog let the model choose a relevant procedure. A file-reading tool completes that mechanism: the model requests a `SKILL.md` file, code reads it, and the instructions enter the next context. Loading a research skill can guide later searches without adding a new search implementation. The instructions can guide reasoning and responses within the next call. Steps that read files, run scripts, or change external state still need execution tools.

Tool discovery with MCP

A registry can start with functions defined in the application. As the system connects to more services, each integration otherwise needs its own code for describing operations, sending requests, and receiving results. Model Context Protocol (MCP) provides a shared protocol for these exchanges. An MCP server exposes tool descriptions and handles requests; the application acts as a client. The server's implementation can evolve independently while the application uses the same protocol to discover and call its tools.

Discovery and execution remain separate. The application obtains tool definitions from a server and selects which descriptions to include in the model's context. When the model requests one of those tools, the application checks the request and routes it to the server, then returns the result to a later call. A large catalog may need search or selective loading for the same reason a skill catalog does: descriptions occupy context.



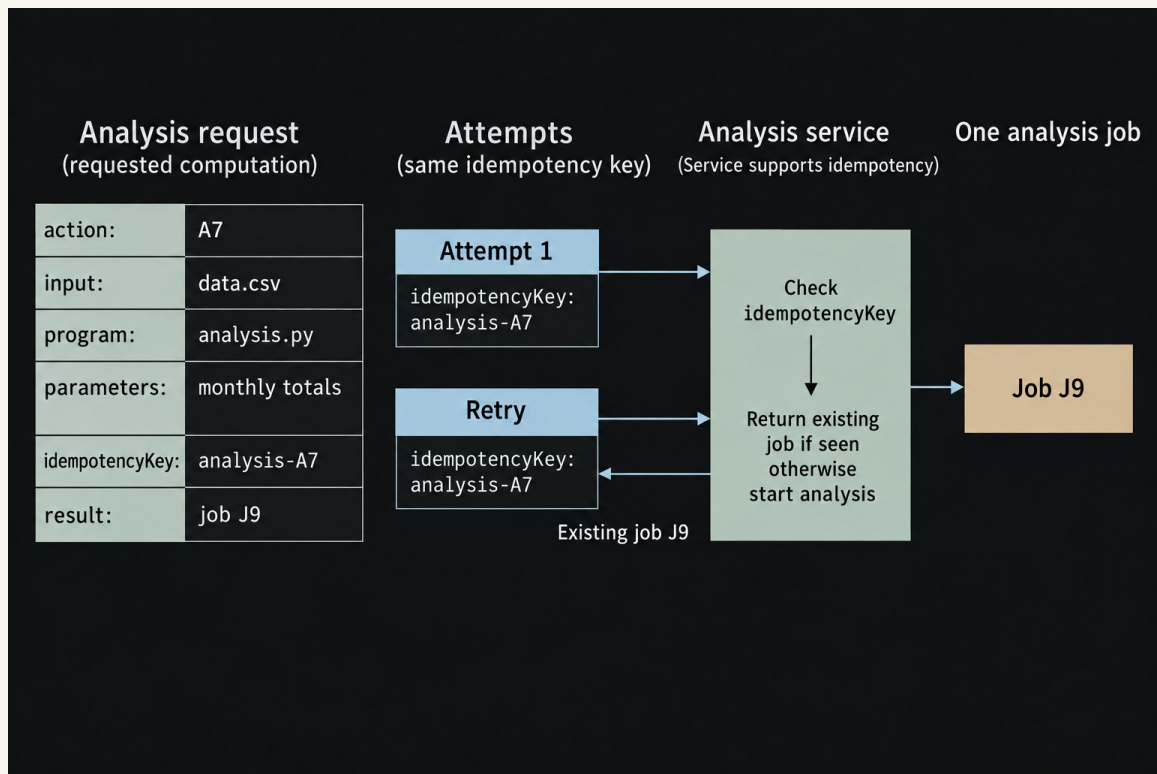
The application supplies tool descriptions, validates requests, and routes calls and results between the model and server.

Execution records

A useful tool result describes what the operation established. A search returns passages and source identifiers; an edit returns the saved revision; a test run returns findings and an exit status. The next call can interpret that evidence and choose a response. The result also needs to distinguish intermediate states: a service accepting a background job does not mean the job has finished. Source identifiers and job identifiers help connect each observation to the operation that produced it.

Sometimes the outcome is unknown. A service can start a data-processing job and then fail to return a response before a timeout. Repeating the request blindly may start the same analysis twice. An idempotency key identifies one logical action across attempts; a service that supports it can recognize the retry and return the existing job instead of creating another. The protection comes from the service's handling of the key, not from merely recording a key beside the request.

An action record connects the model's request, execution attempts, and observed results. For a data-processing job, the record can retain the input files, requested computation, idempotency key, returned job identifier, and output location. The application can use that record to investigate a failure or recover after interruption. It can also select the relevant outcome for the next context, so the model can reason from the observed result.



An idempotency key connects repeated requests to one analysis job when the service supports it.

Agency connects a model decision to an operation and brings the result back as an observation. Repeating that connection lets the next decision depend on what the previous action revealed. The next chapter builds the tool-calling agent loop from these parts, then examines other ways to arrange model calls and computation into larger reasoning structures.

Reasoning

How calls and operations form loops and larger reasoning structures.



An agent loop uses the result of one action to choose the next step.

A reasoning structure combines model calls and ordinary computation to solve a problem. A single call can compare, infer, and plan from the information in its context. When the next decision depends on a search result or the output of a program, the application must execute that operation and supply the result to another call. Agency gives us the connection between a decision and an action. Reasoning concerns how we arrange those decisions and actions into a process.

The Agent Loop

The tool-calling agent loop is the central reasoning structure in an agentic system. The model receives a task and the observations collected so far, then chooses an action. The runtime executes the tool, records the result, and calls the model again with the updated context. The next call can choose another action or return an answer. This repeated cycle lets an agent work toward a result through a sequence of actions whose order need not be known in advance.

The following example handles one action at a time. The helpers represent application code, with an adapter handling the model provider's message format. `assembleContext` supplies the goal and relevant history, preserving each tool request with its result. The record can grow while the next input stays within the context window. `executeTool` validates the arguments, runs the handler, and returns the result as described in Agency.

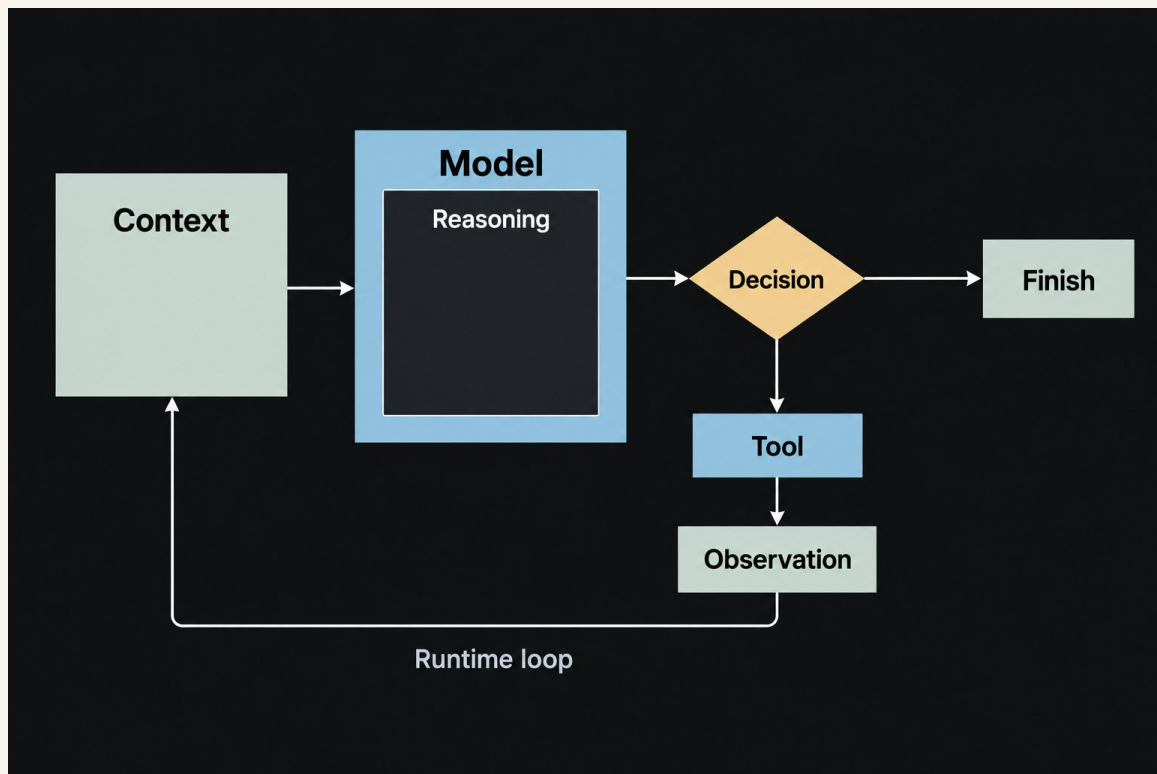
```
const trace = []

for (let step = 0; step < maxSteps; step++) {
  const context = assembleContext({ goal, trace, tools })
  const decision = await callModel(context)

  if (decision.kind === "answer") {
    return { status: "candidate", answer: decision.text }
  }

  const result = await executeTool(decision.toolCall)
  trace.push({ call: decision.toolCall, result })
}

return { status: "stopped", reason: "step limit" }
```



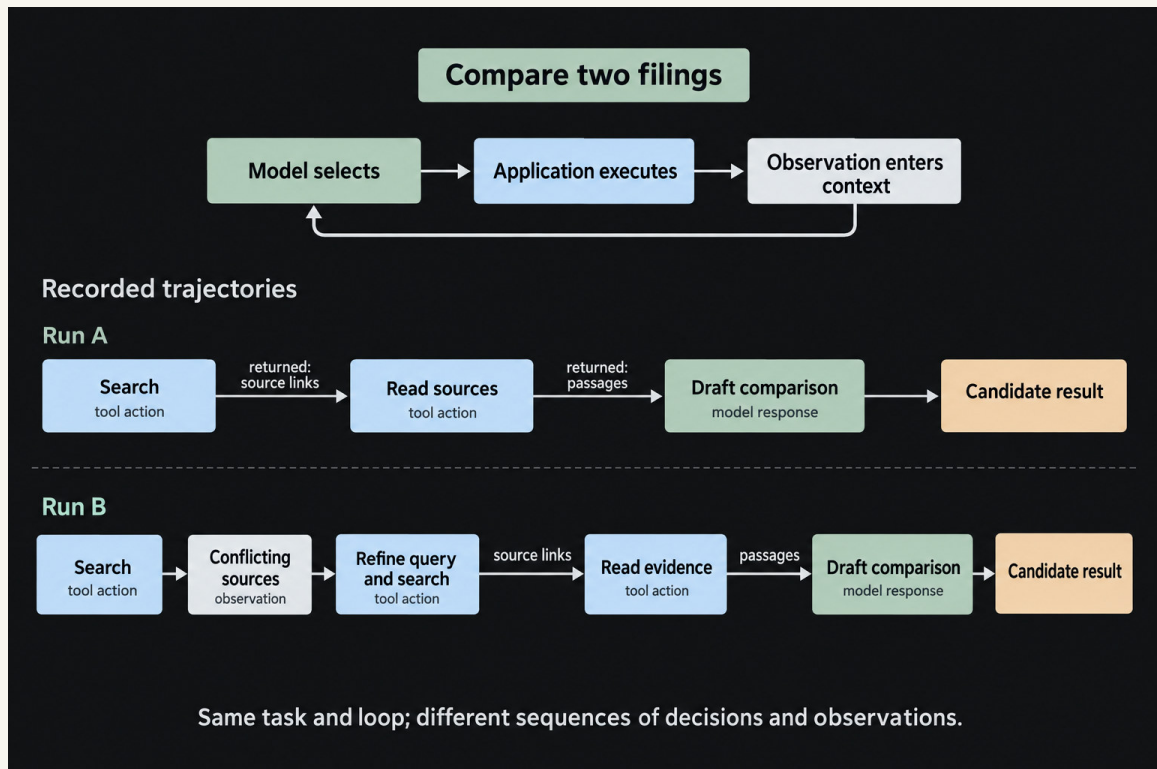
The tool-calling agent loop connects a model decision to execution and observation. Each result supplies information for the next call.

Much of the control flow is implicit in the model's choice of the next action. The application does not need a separate branch for every search result or a predetermined sequence of tools. The model can decide that a claim needs another source, that a failed test requires an edit, or that the available evidence is sufficient. These judgments select the next operation within the same small loop. The requested actions remain explicit and recordable; the application has delegated the choice of their order to the model.

A trajectory is the sequence of decisions and observations produced during a run. Two runs of the same loop can take different trajectories toward the same task. A filing comparison may find the needed sources immediately and proceed to a draft. Conflicting sources may instead lead to a narrower query and more reading before drafting. The runtime repeats the same cycle in both cases; the observations and the model's interpretation determine which actions follow.

The loop aims to converge on a result: successive actions obtain missing information, resolve difficulties, and complete the task. Repetition alone does

not guarantee that progress. An agent can revisit the same sources or stop before resolving an important question. The code therefore distinguishes a candidate answer from a run stopped by its step limit. Acceptance may still require a test, a review, or confirmation that an intended change occurred. Deadlines and cancellation can also stop execution without completing the task.



The runtime repeats the same cycle. Model decisions and returned observations determine the sequence of actions; either run still produces a candidate that may need checking.

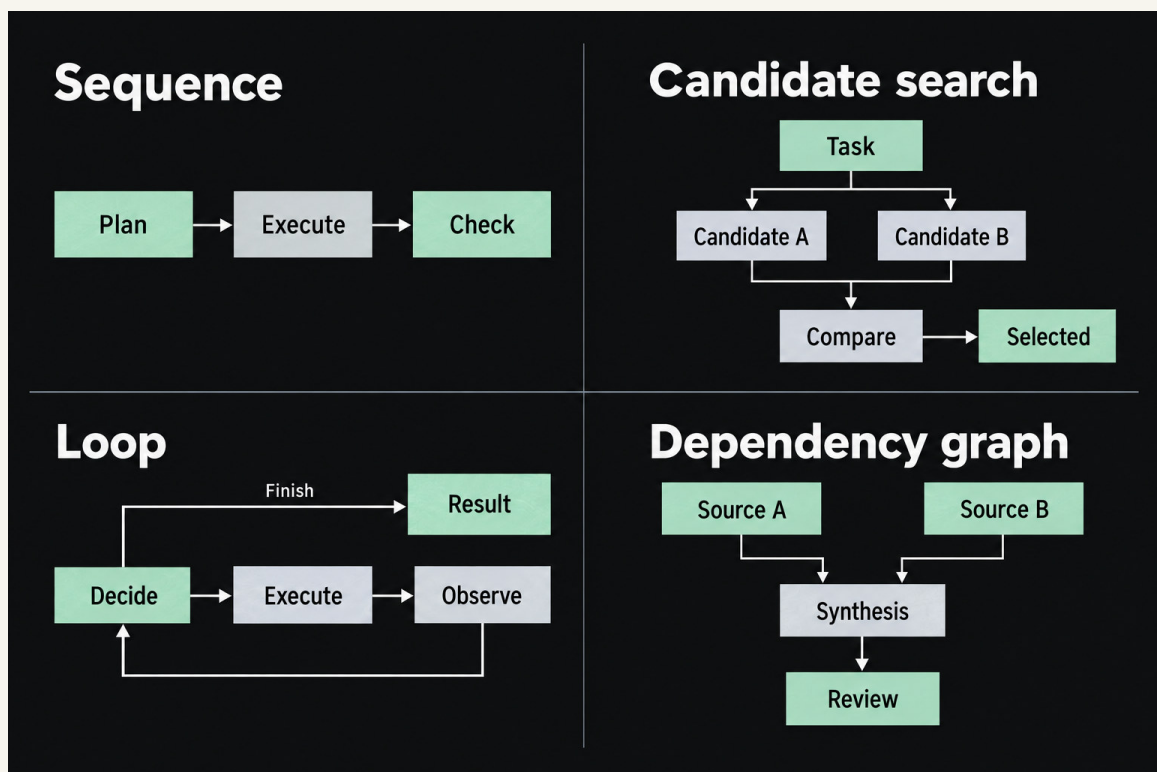
Composition

The agent loop is one way to compose calls. When the steps are already known, a sequence can be simpler: retrieve a document, extract the relevant passages, then write a summary. Each stage receives a defined input and produces something the next stage needs. Code chooses the order, so the model does not have to rediscover the procedure on every task.

Other arrangements support work that branches or has several dependencies:

- **Candidate search** explores several approaches before choosing one to continue. A tree can represent candidates that branch into further alternatives.
- A **directed acyclic graph (DAG)** expresses dependencies without a cycle. Two source sets can be examined separately, with synthesis waiting for both.
- A **routing branch** selects one path, such as sending a calculation to a code-execution step.

The choice follows how much of the process can be specified in advance. A pipeline fixes the stages; a tool-calling loop leaves the next operation open to model judgment. A graph combines dependent and independent stages. Any stage can itself contain a smaller structure, so a fixed research pipeline can include a search loop. An agent can also invoke a tool whose implementation runs a fixed pipeline. Composition lets us combine a known procedure with decisions that depend on new observations.

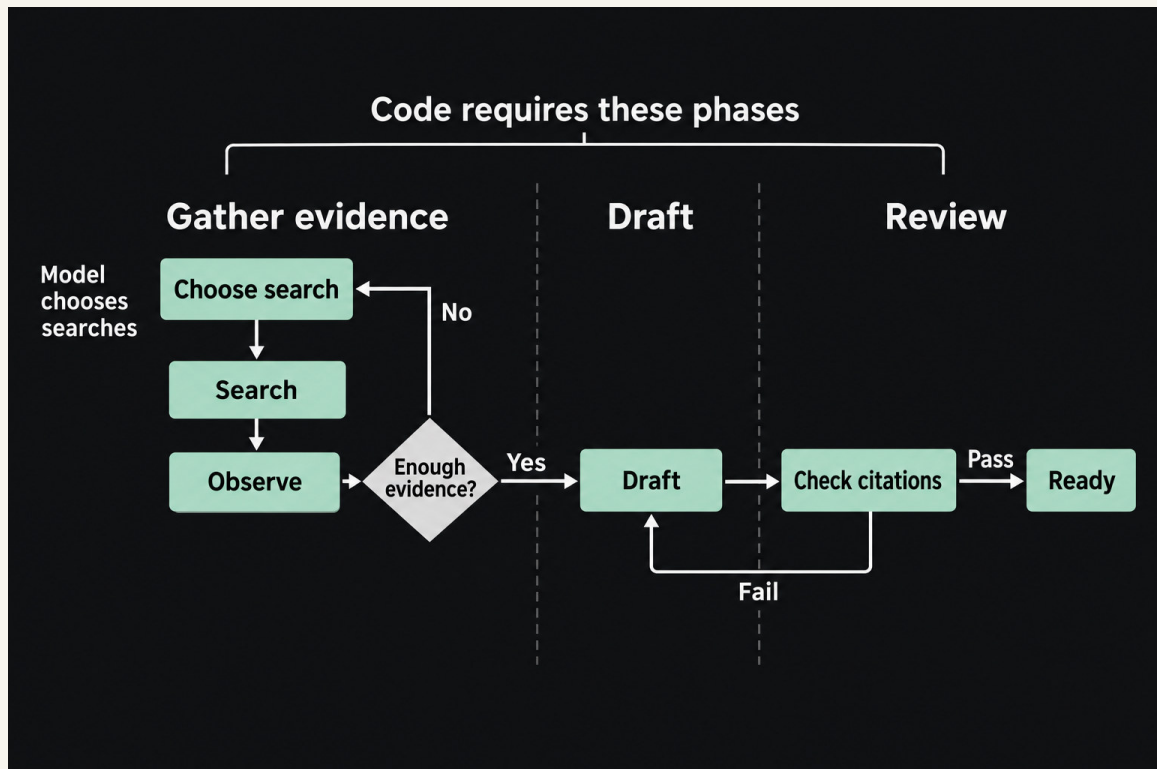


A sequence fixes an order, candidate search compares alternatives, a loop responds to observations, and a dependency graph waits for required inputs.

Required stages

Some steps need to happen on every run. If every draft must pass a citation check, code can require that stage. The model can still choose searches based on what it finds in the sources. Describing a check in a prompt may lead the model to perform it; putting the check in the program ensures it runs before the draft can be accepted.

For example, a process can have three required phases: gather evidence, draft, and review. Within the first phase, a model chooses searches and examines their results. Code controls when the process may advance to drafting. A failed review can return the draft for revision. The inner loop discovers a useful sequence of actions while the surrounding workflow preserves the required stages.



A search produces an observation. The workflow either seeks more evidence or advances to drafting, then checks the draft.

Operations between model calls

Every return from a model call gives the runtime an opportunity to change how the process continues. After a search, code can retrieve related evidence, select the passages needed for the next decision, summarize older exchanges to make room, or choose a model for the next step. These operations determine what the next invocation can use. A continuing agent retains its task through reconstructed context, not through a model carrying private state from one call to the next.

The table below names six common operations that code can perform between calls. Each changes what the next call receives or how the work continues.

Operation	Concrete mechanism
Inject	Await a search result and include its evidence in the next input.
Verify	Check citations and use the findings to choose the next step.
Focus	Select one claim and the passages needed to review it.
Branch	Select one path with a condition, or start several paths at once.
Merge	Collect the required branch results, preserve disagreements, and synthesize.
Loop	Generate another candidate with feedback until accepted or stopped.

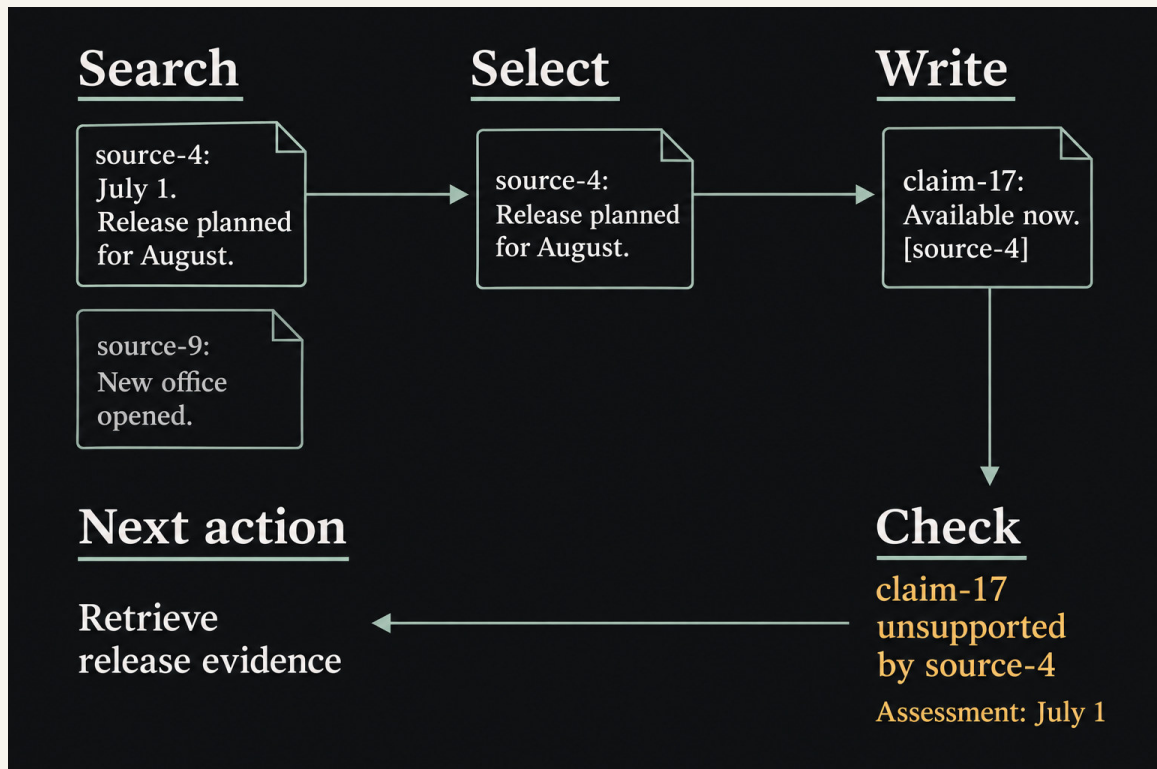
A short TypeScript sequence makes three of these operations visible. Search obtains information, selection constructs the writer's input, and validation produces a finding for the controller. The helpers stand for application code. The next control decision can accept the draft, revise it with the finding, or stop.

```

const sources = await search(question)           // inject
const evidence = selectPassages(sources, question) // focus
const draft = await write({ question, evidence })
const check = verifyCitations(draft, evidence)    // verify

await record({ draft, evidence, check })
// The controller must now choose accept, revise, or stop.

```



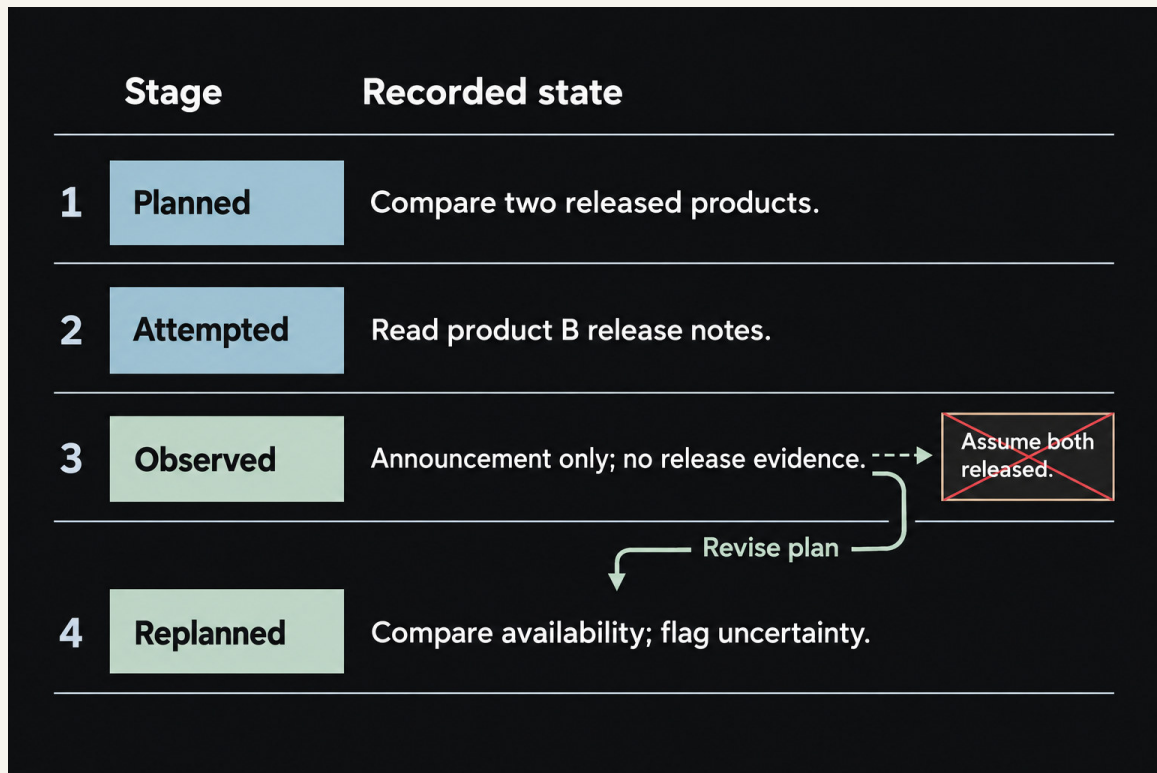
Search, passage selection, drafting, and checking form one process. The citation finding gives the controller a concrete reason to revise or gather more evidence.

Planning

An agent loop does not require a separate planning stage. Choosing the next action can include deciding what must be learned or accomplished first. An explicit plan becomes useful when the task has dependencies or unfinished work that later calls need to recover. The plan records an intended sequence; observations can change that sequence as the agent works.

For example, a plan to compare two released products rests on the assumption that both have shipped. If a search finds only an announcement for one, the next call may seek release evidence, explain the uncertainty in the comparison, or ask

whether to narrow the task. A useful plan preserves what was intended alongside what was attempted and observed. Later calls can revise the remaining work instead of continuing as though the original assumption had been confirmed.



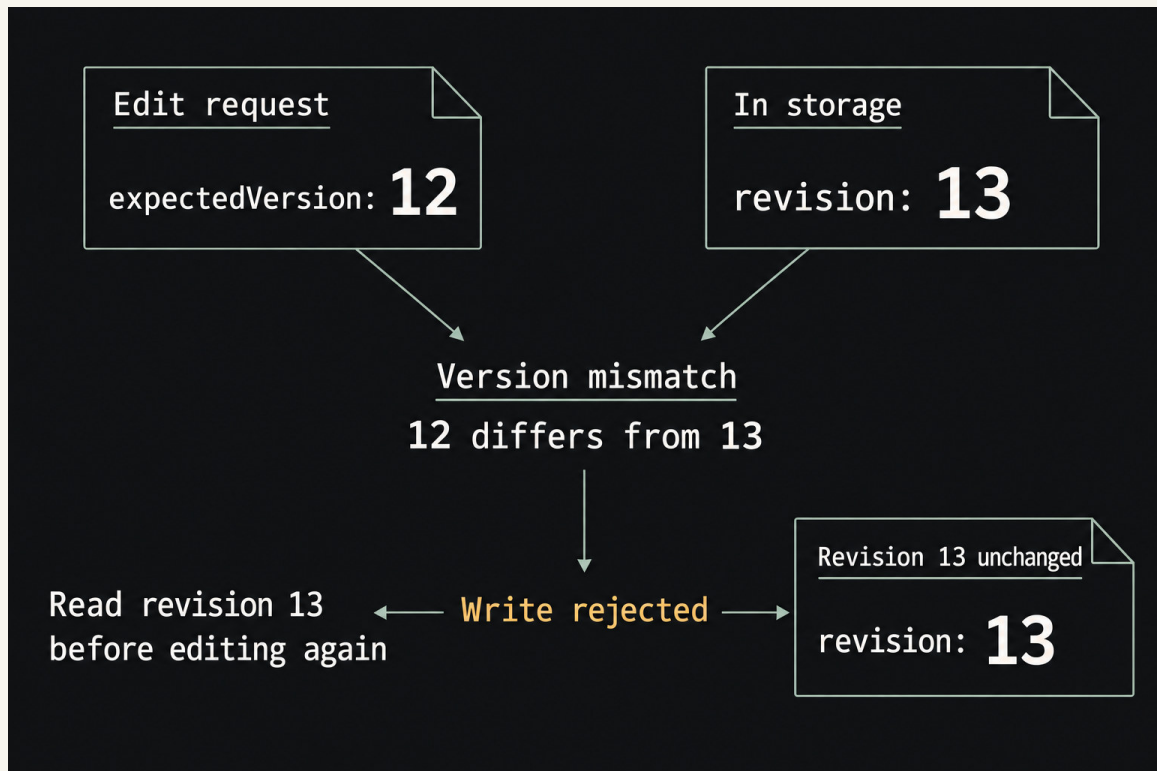
A plan is useful when observations can change it. The runner must preserve what happened and reconsider the next step.

Execution traces

A trace records the trajectory the system actually took: the input to each decision, the proposed action, the observed result, and whether the run continued or stopped. That record makes implicit choices inspectable after execution. Repeated searches may reveal that the model lacked a useful observation or failed to use one it had. A rejected edit may reveal stale document context. The repair depends on which part of the process failed.

Additional calls make sense when they add an observation, a different view of the problem, or a useful check. They also add latency and opportunities to lose information. A pipeline can guarantee that a stage runs, but cannot guarantee

that the stage reaches a correct conclusion. The useful design question is what each additional step contributes toward the required result.



The trace establishes a stale write attempt: the request expected revision 12, storage held revision 13, and the rejected write left revision 13 intact. The next step is to read the current version.

The agent loop turns model decisions and tool results into a continuing attempt to solve a task. Composition lets us place that loop within larger procedures, add independent investigations, and combine their findings. Once several reasoning structures have distinct responsibilities, we need to decide what each receives, when each runs, and how their results fit together. Those are the concerns of coordination.

Coordination

How agents divide work and combine their results.



Separate agents work with different contexts and return contributions that the system must combine.

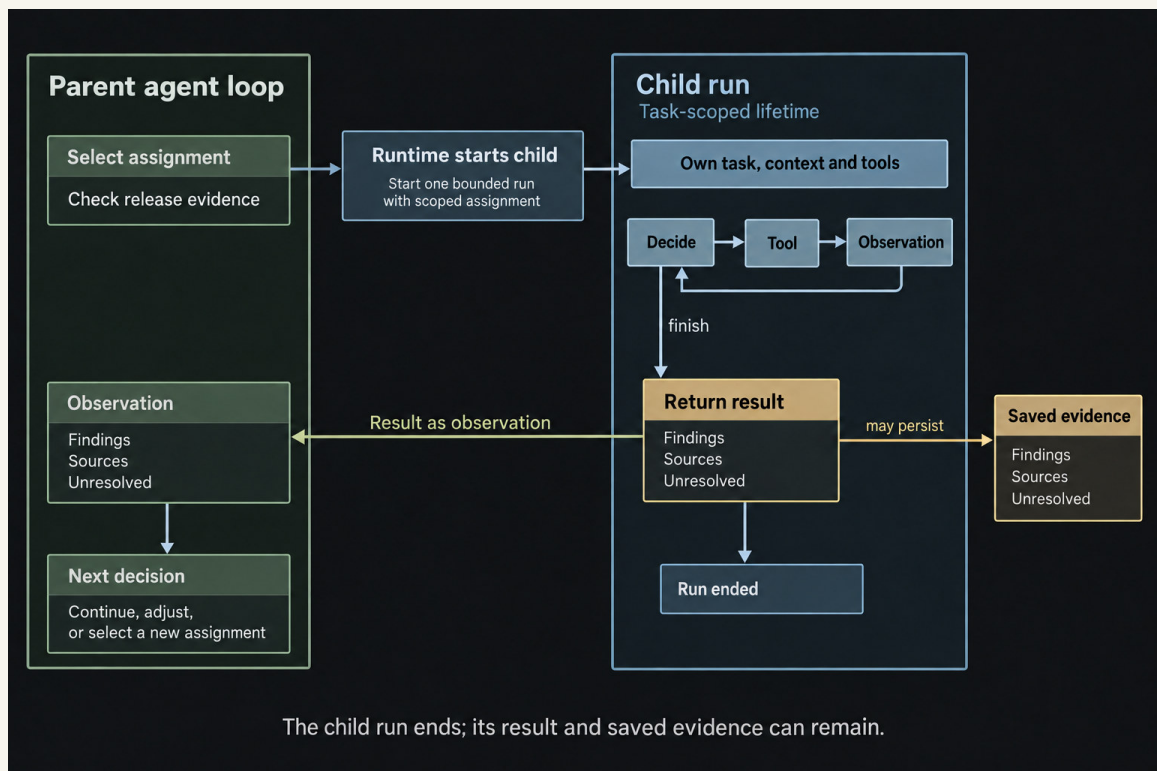
An agent loop can keep taking actions, but each model call still has a limited context window. Instructions, source passages, tool results, and earlier decisions all occupy space in that input. As the work grows, the application must select or summarize what the next call receives. Memory preserves material for later retrieval, but each call still receives only what fits in its context. These techniques can keep one loop effective, but some parts of a task need extensive investigation whose intermediate details matter little to the rest of the work.

A product comparison, for example, may need to establish whether one product has actually shipped. Resolving that question could require many searches and conflicting documents. The comparison needs the finding, its evidence, and any remaining uncertainty. A separate agent can conduct the investigation in its own

context and return those results, keeping the intermediate work out of the parent's input. The first design question is which parts of a task allow this separation: what information does each part need, and what must it return for the larger task to continue? Coordination concerns how we divide that work and bring its results together.

The Subagent Pattern

A subagent is an agent run created for a specific assignment from another agent. The parent supplies the assignment, and the application starts a child with its own context and tools. The child works through its own tool-calling loop, returns a result or a reason it stopped, then ends its run. A saved record or file can outlive that run; a new operating-system process is not required to create a subagent.



The parent delegates an assignment and receives the child's selected findings as an observation. The child run ends; its result and saved evidence can remain.

Delegation fits directly into the agent loop from the previous chapter. The parent requests a tool such as `delegate`, and the tool's implementation runs the child loop. The child's result becomes an observation in the parent's next context. The

example below uses application interfaces. `createChild` constructs the child's context and execution limits from the assignment, the current parent run, and the configured tools and workspace. `runAgent` executes that bounded loop and returns a candidate result or a stopped status.

```
// parentRun and investigationConfig come from the application.
async function delegate(request, parentRun) {
  const child = await runtime.createChild({
    parentRun,
    configuration: investigationConfig,
    assignment: request.assignment,
    sourceIds: request.sourceIds,
  })

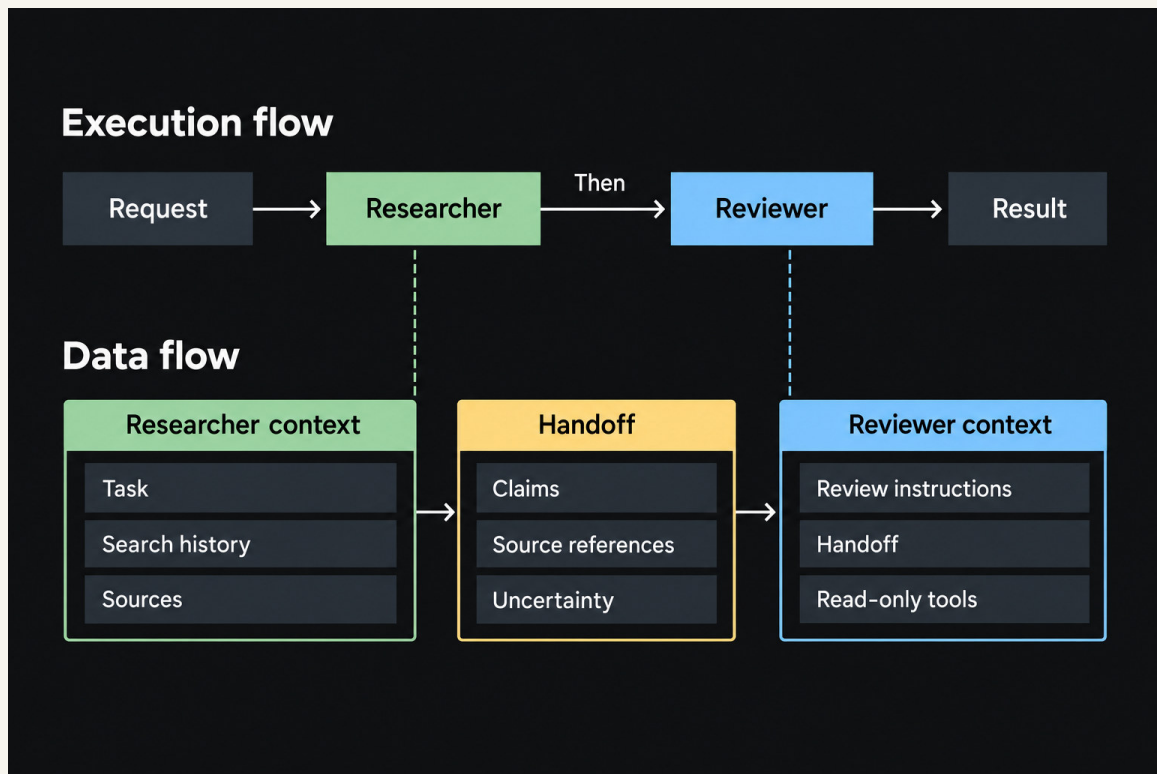
  return runAgent(child)
}

// The parent dispatcher handles a validated delegation request:
const result = await delegate(call.arguments, parentRun)
trace.push({ call, result })
// The parent's next context includes the child's returned outcome.
```

Task decomposition

A useful boundary separates work that needs substantial local detail but has a clear result for the larger task. “Establish product B's release status, with sources and unresolved questions” gives the child a result to pursue. The assignment must also supply product identifiers, relevant constraints, and what counts as “released.” A role name such as “researcher” leaves those requirements unspecified. If every step requires the parent's latest conclusions, or the parent must replay the whole investigation to use the answer, keeping the work in one loop may be simpler. Delegation earns its cost when a focused assignment and usable return save more context or time than the handoffs consume.

For each assignment, we need to decide when the subagent runs and what information it receives. A reviewer can start after an investigation finishes, yet still lack the evidence needed to review it. Copying the entire investigation history preserves more material but may bury the claims under search attempts. Selecting claims, supporting passages, and uncertainties gives the reviewer a focused input. The order of execution and the information passed between agents both affect whether the reviewer can do its job.



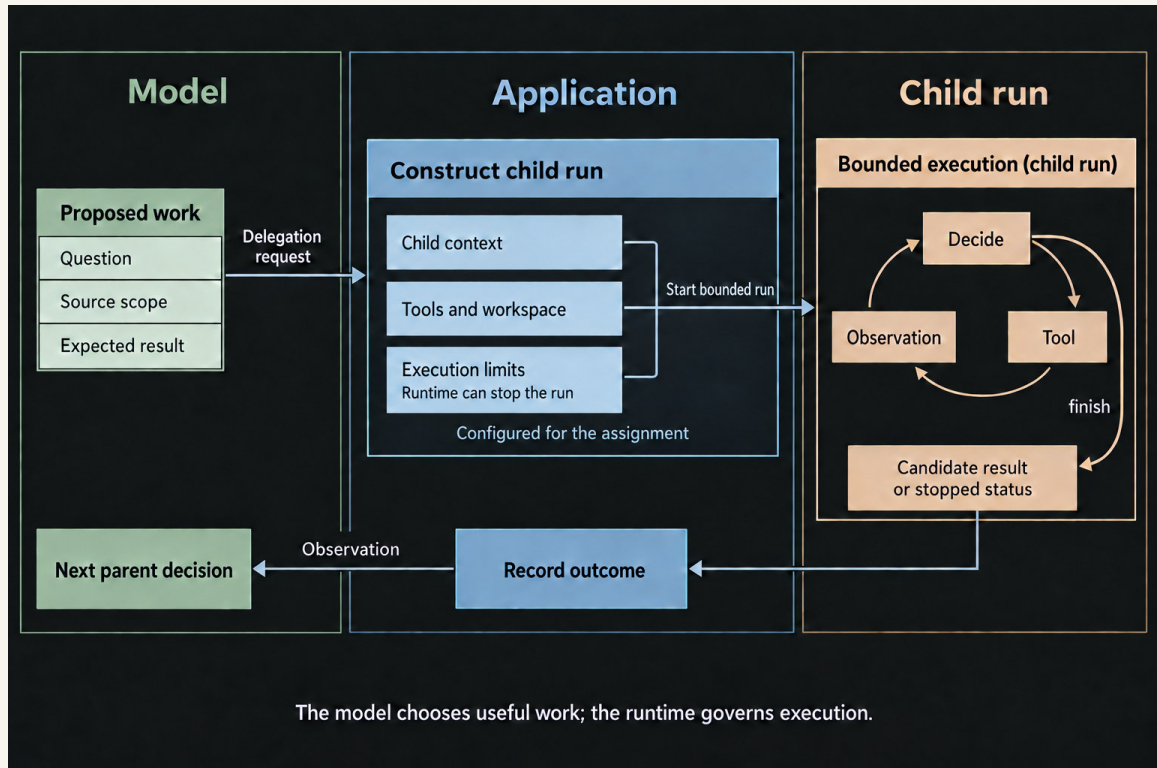
Execution flow determines when reasoning structures run. Data flow determines what each receives from the others.

Model-directed delegation

An orchestrator is an agent that manages assignments and combines their results. Its model can decide whether a task benefits from a separate investigation, formulate the question, and request another investigation after reading a result. A source conflict might lead to a narrowly scoped follow-up; sufficient evidence might lead directly to writing. As with the agent loop, the runtime supplies the repeated structure while model decisions determine the path through it. The full collection of subtasks need not be known before work starts.

The application sets up the child's environment: the source material, tools, and workspace it can use, along with limits on time or number of steps. These choices determine what the child can inspect and do. Separate contexts do not isolate shared files, so the application must also decide whether workers share a workspace or write separate outputs.

A known procedure can fix some of the decomposition. A comparison service may always investigate two source collections and require a review before accepting the synthesis. Within those stages, a model can choose narrower questions or additional searches. This combines predictable requirements with adaptable investigation. Further delegation can repeat the subagent pattern inside a child run, provided the runtime carries the scope and limits through to its descendants.



The model proposes work. The application constructs the child's context, tools, workspace, and execution limits, then returns the outcome to the parent.

Parallel execution

Dependencies determine which assignments can usefully run together. An investigation of company filings and an investigation of product documentation may start independently. Their synthesis needs both findings, and a review of the synthesis needs the draft. That task therefore has parallel investigations followed by dependent stages. If one investigation discovers information the other requires, the orchestrator must pass that information on or revise the assignments. Parallel execution is useful when the work is sufficiently independent and combining it is inexpensive.

A join collects the outcomes of separate runs before deciding how to continue. In TypeScript, `Promise.allSettled` waits for each run to return or throw an error. A fulfilled promise can still contain a child that stopped early, so the code preserves the child's status. The illustrative `assessCoverage` check compares the returned evidence and unresolved questions with the information needed for synthesis.

```
const settled = await Promise.allSettled(
  assignments.map(request => delegate(request, parentRun)),
)

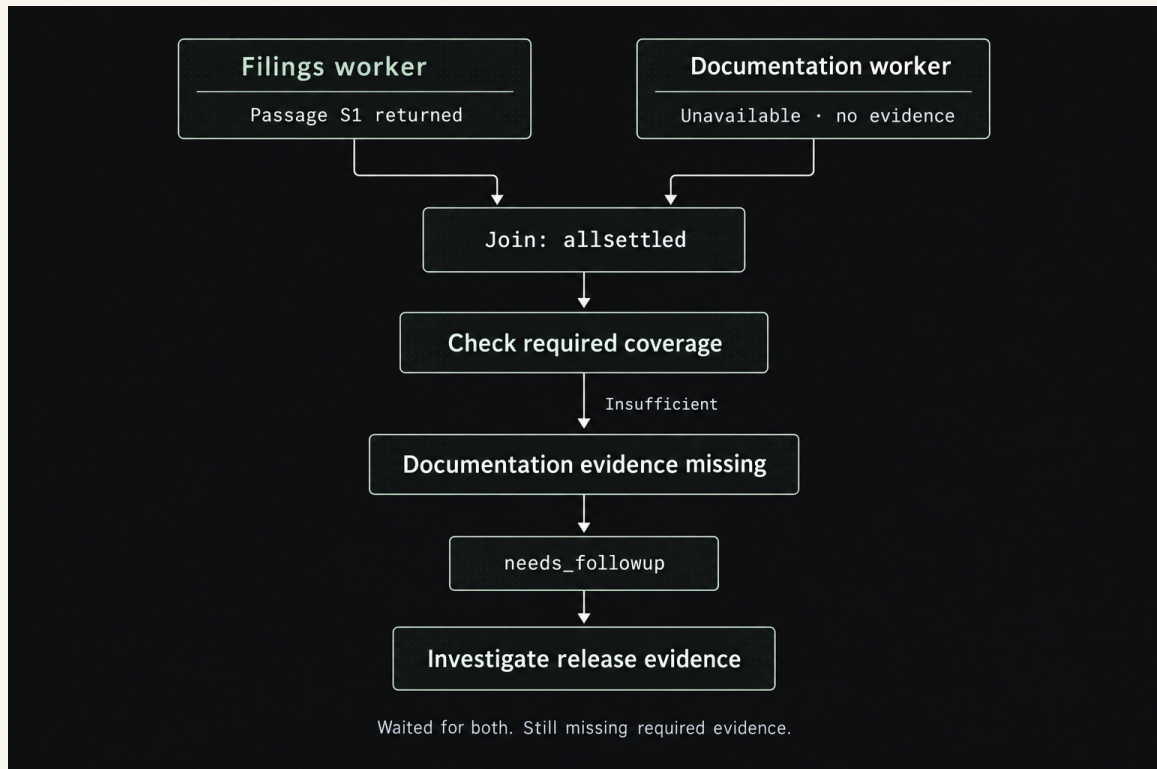
const outcomes = settled.map((entry, i) => ({
  assignmentId: assignments[i].id,
  result: entry.status === "fulfilled"
    ? entry.value
    : { status: "run_error", reason: "No child result returned" },
}))

const coverage = await assessCoverage({ goal, outcomes })
if (!coverage.sufficient) {
  return { status: "needs_followup", outcomes, coverage }
}

return {
  status: "candidate",
  answer: await synthesize({ goal, outcomes }),
}
```

The first delegation example waits for a child before the parent continues. An asynchronous interface can instead return a run identifier, allowing the parent to do other work and collect the result later. The application chooses how to continue based on what the next stage needs:

- **All required results.** Synthesis waits for findings from every required investigation. An unavailable source remains missing coverage.
- **The first adequate answer.** When several runs try to satisfy the same requirement, one sufficient result can let the parent continue.
- **Results as they arrive.** The parent reads each finding while other investigations continue, so it can act on useful information sooner.



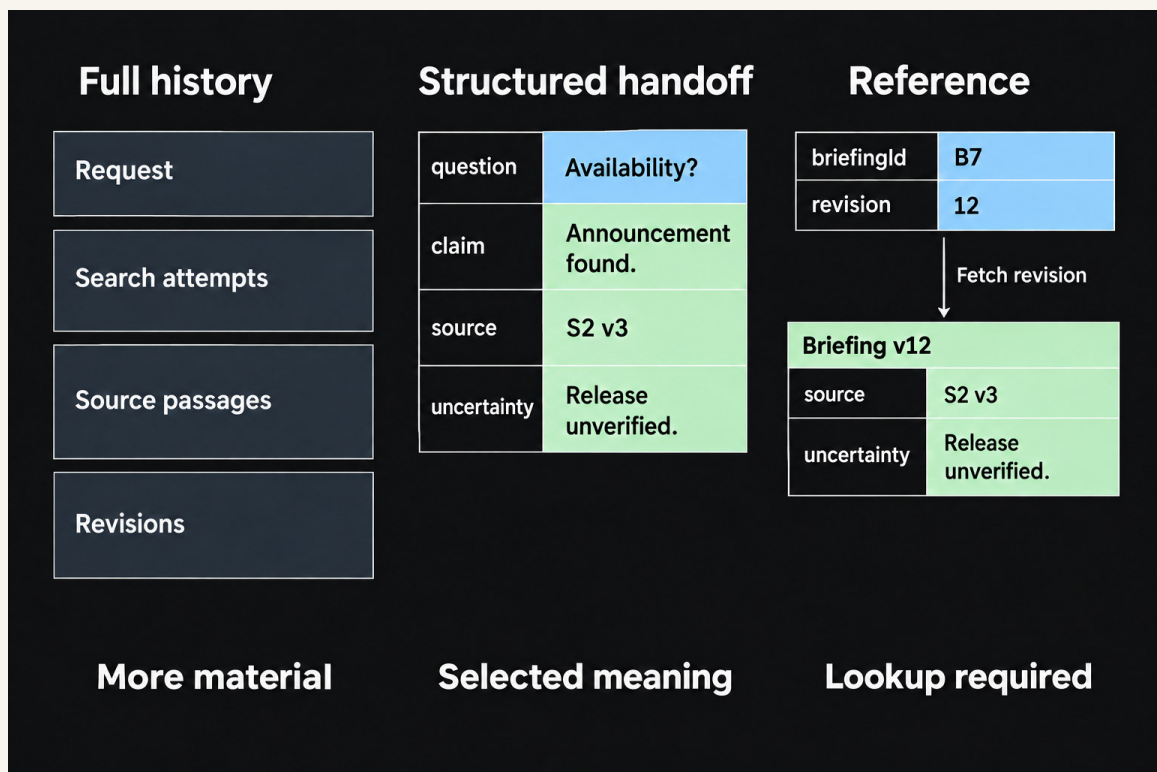
The join collects both outcomes. Because the documentation evidence is missing, the coverage check returns `needs_followup` rather than a synthesis.

Results, verification, and recombination

A returned result needs enough detail for the parent's next decision. “Research complete” reports a status without providing the findings. A useful release-status result identifies the product, separates an announcement from evidence of availability, cites the supporting passages, and preserves unresolved questions. The parent can then compare findings, inspect a source, or commission another bounded investigation.

Combining results requires interpreting them. Two workers may have examined different dates or used different meanings of “released.” The parent must resolve those differences before combining their claims. Code can check for missing fields or assignments that returned no result. Judging whether the evidence supports a conclusion may need another model call or a person's review. When the evidence is insufficient, the next step is a more precise assignment or an explicit statement of the gap.

The result can travel as text, a structured summary, or a reference to saved work. Full history carries more detail at a higher context cost. A summary selects meaning and can lose a qualification. A file or document reference keeps the return small, but the receiver must be able to fetch the relevant version. The representation should follow what the next decision needs. Source references and remaining uncertainty help the parent recover detail without loading every step the child took.



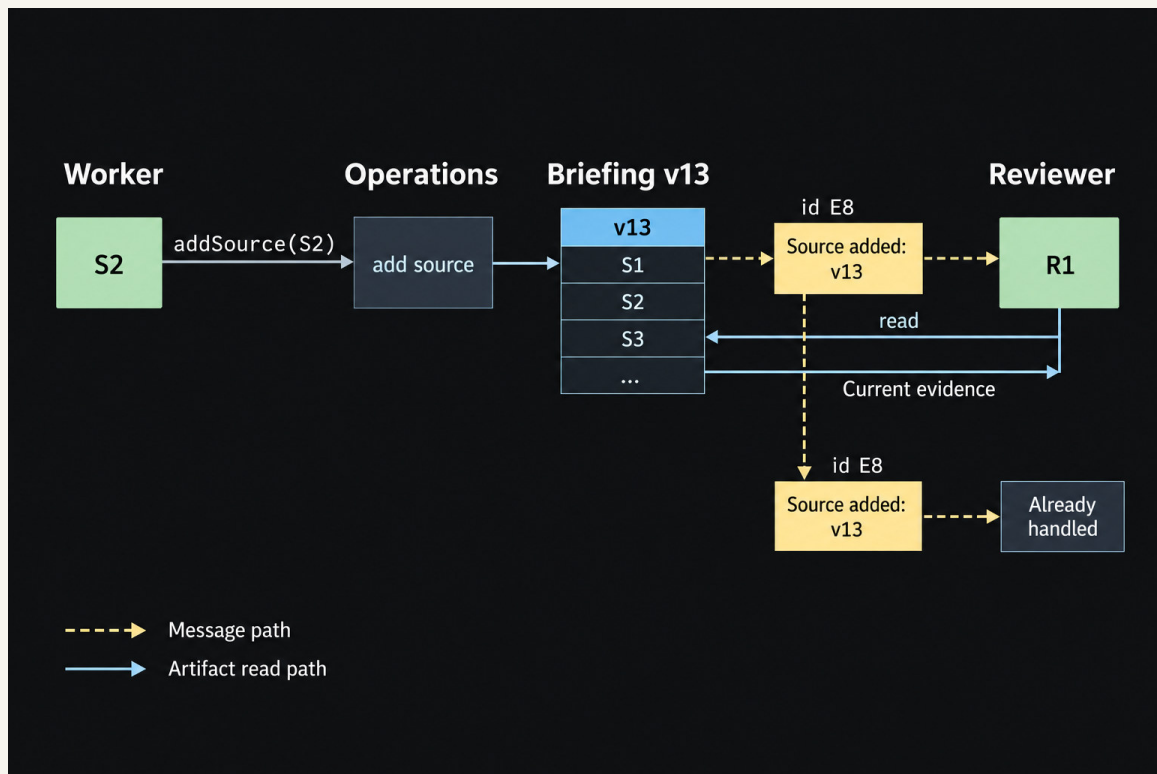
Full history, a structured summary, and a reference preserve different things. A reference is useful only if the receiver can resolve it.

Shared workspaces

Subagents can leave work in an environment that survives their runs. An investigator might write `research/product-b.md` and return its path with a short account of the findings. The parent reads the note when it needs the evidence. Each worker can focus on its assignment while files or database records provide a common place for results. The workers need no direct conversation with each other when the parent and shared workspace supply the necessary connections.

A message can announce that a document changed while the document holds the actual work. If an investigator reports that revision 13 contains a new source, a reviewer can read that revision before continuing. This separates notification from evidence: the message directs attention, and the read supplies the material for the next decision. Shared state can therefore support coordination alongside returned messages.

Shared access still needs ownership. Two subagents can have separate contexts while editing the same file and overwriting each other's changes. Separate output files, a single parent that integrates proposals, or version-checked updates can control that interference. The environment's operations determine which changes can safely coexist. The next chapter develops these artifacts and operations as parts of the system in their own right.



Notifications and artifacts can cooperate. The stored version remains the reference for the work, and updates still need a conflict policy.

The subagent pattern lets a model turn a broad task into bounded assignments, inspect their outcomes, and adapt the remaining work. Context limits explain why information needs to be distributed; dependencies explain when results must come together. The runtime makes those boundaries executable while the model decides how to use them. The work can outlive any particular run through the artifacts those runs produce.

Artifacts

How shared objects preserve work and organize further contributions.



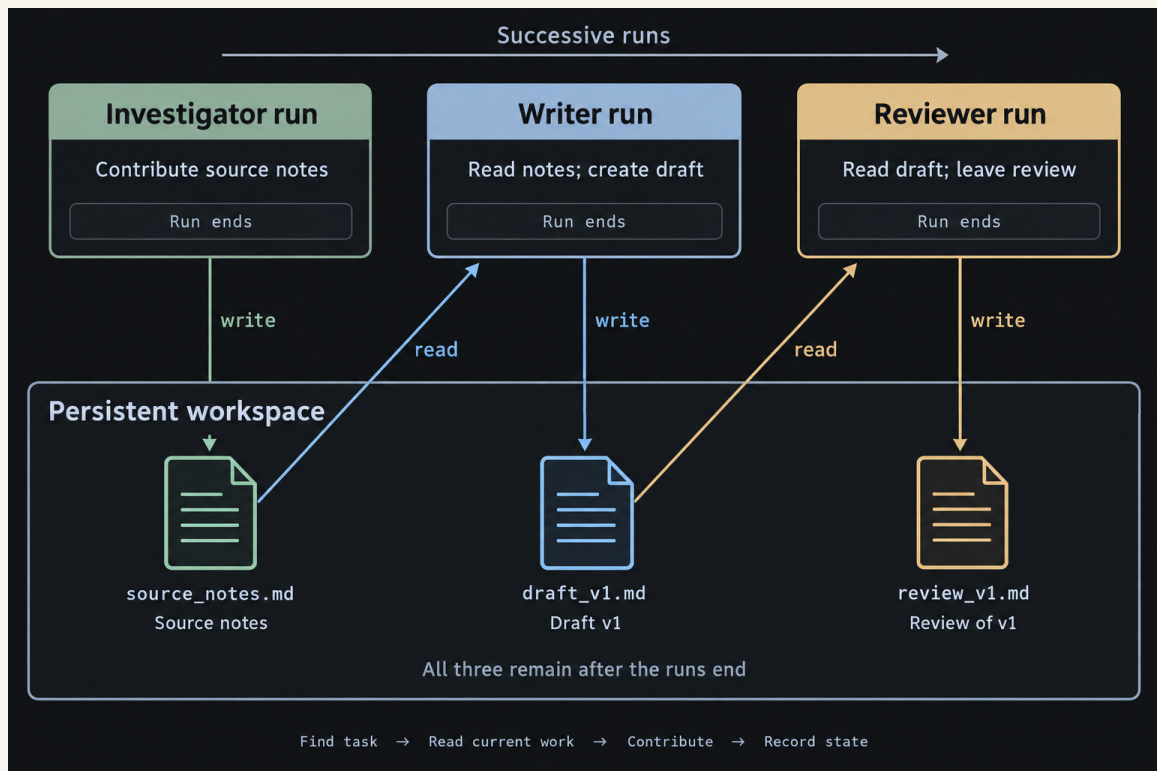
Shared artifacts preserve contributions and give later work something concrete to build on.

Coordination lets several agents contribute to a task, each with its own context and a bounded run. Their work needs to survive those runs and become available to the next participant. An artifact is an object produced by work and saved for later use. Documents, source files, datasets, and task records give the system something to inspect, extend, and use after the agents that created them have stopped.

Research notes can supply evidence for a draft; the draft can receive a review; the review can guide a revision. Each contribution leaves material for the next step. An agent reads the parts relevant to its assignment and saves its contribution in the shared workspace. A path or identifier connects the assignment to the work, while a revision identifies the version examined. The

project can grow beyond any one context without requiring every participant to reconstruct every preceding investigation.

Persistence gives these contributions a place to accumulate. Their organization determines how another agent can participate. A task record can identify the result still needed, its current owner, and whether it is ready for review. Instructions and operations associated with that record explain how to claim the task, contribute a result, and submit it for review. The environment can therefore preserve both the work and a protocol for continuing it.



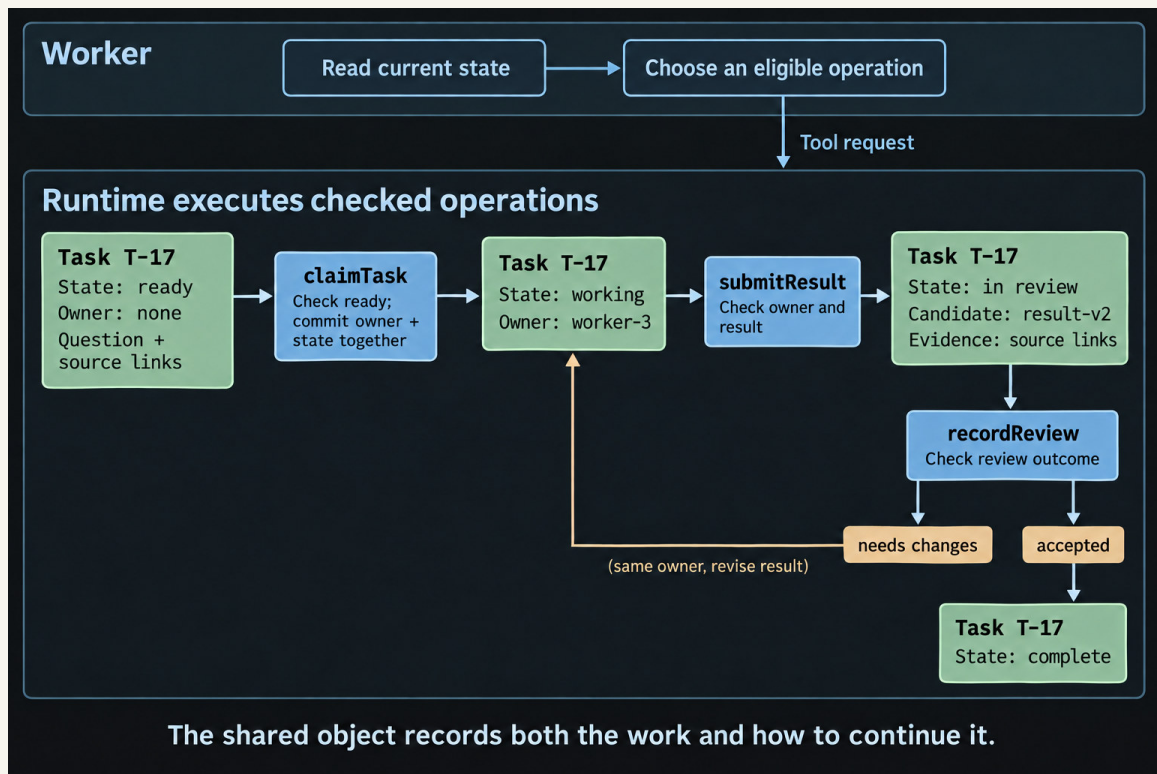
Each bounded run leaves work that another run can read and extend. Notes, drafts, and reviews remain in the workspace after their creators finish.

Types and protocols

An artifact type describes a kind of work object and the interactions it supports. A document has sections that can be edited; a dataset has records that can be queried; a task has an assignment and a state that can change. The type determines which parts of the work are identifiable and which operations make sense. Its interface exposes those parts and operations to people, agents, and other software.

A protocol describes how participants use that interface over time. A research task might move from ready to working when a worker claims it, then to in review when the worker submits findings and evidence. A reviewer can accept the result or request changes. The task's current state tells each participant which contribution is needed next. The model still decides how to investigate the question or judge the evidence, while the shared protocol supplies the procedure for handing work on.

This moves recurring instructions and bookkeeping into the environment. A parent agent does not need to repeat the review procedure for every assignment if the worker can find it through the task's interface. Some protocols are conventions that people and models interpret; others have operations that check conditions and update related state. A Markdown instruction can describe who should claim work. An implemented claim operation can also prevent two workers from acquiring the same task. The design must distinguish the guidance participants read from the rules software enforces.



Workers inspect state and choose an operation. Checked operations claim work, submit a candidate, and record a review so later participants can determine what comes next.

Artifact storage

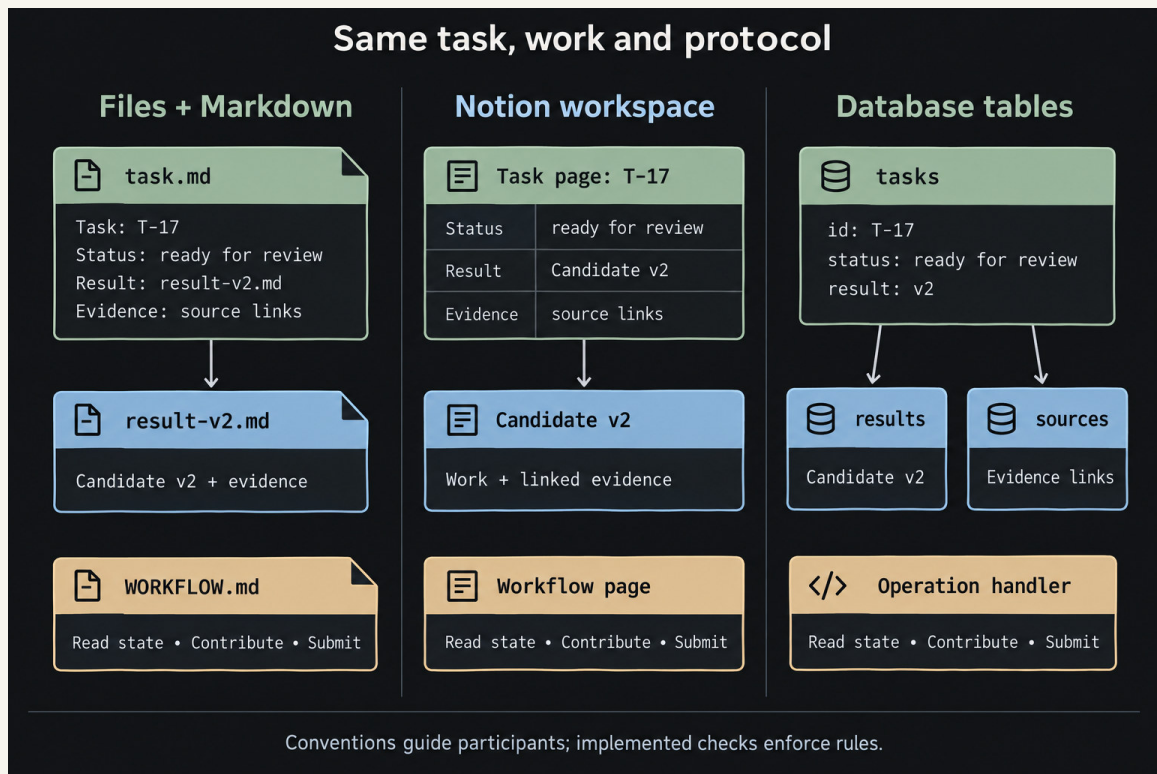
A filesystem can express an artifact type through folders, document templates, and shared instructions. Task files can link to findings under `results/`, evidence under `sources/`, and reviews under `reviews/`. A `WORKFLOW.md` file can describe the allowed status changes and what each contribution must contain. The example task points to that protocol; all paths are relative to the workspace root. The model interprets these conventions when it reads and edits the files, using search, diffs, and version control to inspect the work.

```
# tasks/T-17.md
Question: Has product B shipped?
Protocol: WORKFLOW.md
Status: in_review
Owner: worker-3
Candidate: results/T-17-v2.md
Evidence: sources/release-notes.md
Review: reviews/T-17-v2.md

# WORKFLOW.md
Submit findings with their evidence and a versioned candidate.
Review the linked candidate and record acceptance or requested changes.
A revised candidate needs a new review.
Mark the task complete only after acceptance.
```

A Notion workspace can express the same design with task pages, status and owner properties, links to results, and relations to sources. A filtered view can expose tasks awaiting review, while a workflow page explains the review procedure. The saved properties and instructions make the work understandable to people and agents using the workspace. Additional tool handlers or automation can implement checks that ordinary page edits would leave to the participants.

Database tables can represent those tasks, results, and sources as rows with identifiers and relationships. Queries select eligible work, and application operations perform the required updates. A filesystem, a Notion workspace, and a database can all support the same protocol; the choice depends on how participants discover, inspect, and change the work. The representation needs enough structure for those operations, and its interface must make that structure accessible to the agent.



Files, a Notion workspace, and database records can represent the same task and procedure. Conventions guide participants; implemented checks enforce particular rules.

Coordination through artifacts

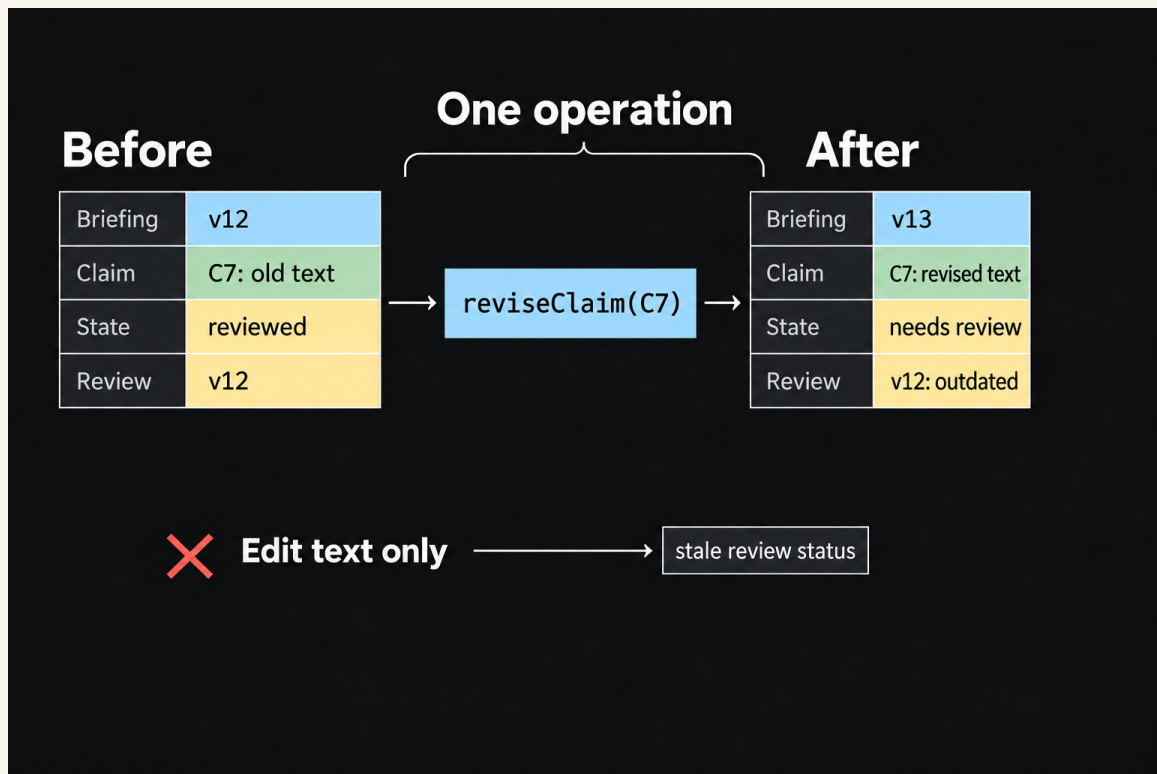
Shared state can carry part of the coordination that a parent would otherwise direct. A worker discovers a ready task, claims it, and later submits a candidate result. A reviewer discovers that candidate through the task's new state. The participants need a way to discover changes, such as a query or notification, but each handoff can refer to the stored task and its protocol. The parent's remaining work can focus on priorities, difficult dependencies, and results that need interpretation.

An operation turns a protocol step into code. A task may have been claimed since a worker read the ready list, so the claim operation must check that it is still available when recording ownership. The illustrative handler below uses a conditional update that checks the status and revision and records ownership in one indivisible storage operation. The revision comes from the selected task, and the worker identity comes from the runtime. A successful claim establishes ownership; the investigation and its review still remain.

```
// Application interface backed by an atomic conditional update.
async function claimTask(candidate, run) {
  const claimed = await taskStore.updateWhere({
    id: candidate.id,
    status: "ready",
    revision: candidate.revision,
  }, {
    status: "working",
    owner: run.workerId,
    revision: candidate.revision + 1,
  })

  return claimed
    ? { status: "claimed", task: claimed }
    : { status: "conflict" }
}
// After a conflict, refresh the available work before choosing again.
// Additional prerequisites need checks under the same protection.
```

Operations can also maintain relationships that would otherwise require several coordinated edits. If a reviewed claim changes, `reviseClaim` can save the new content, advance the revision, and mark the previous review as outdated together. The model proposes the revised claim; the handler updates its revision and review status. The review remains a record of the version it examined, while the new version needs its own assessment.



A domain operation can update content, advance its revision, and mark the previous review as outdated together.

External computation

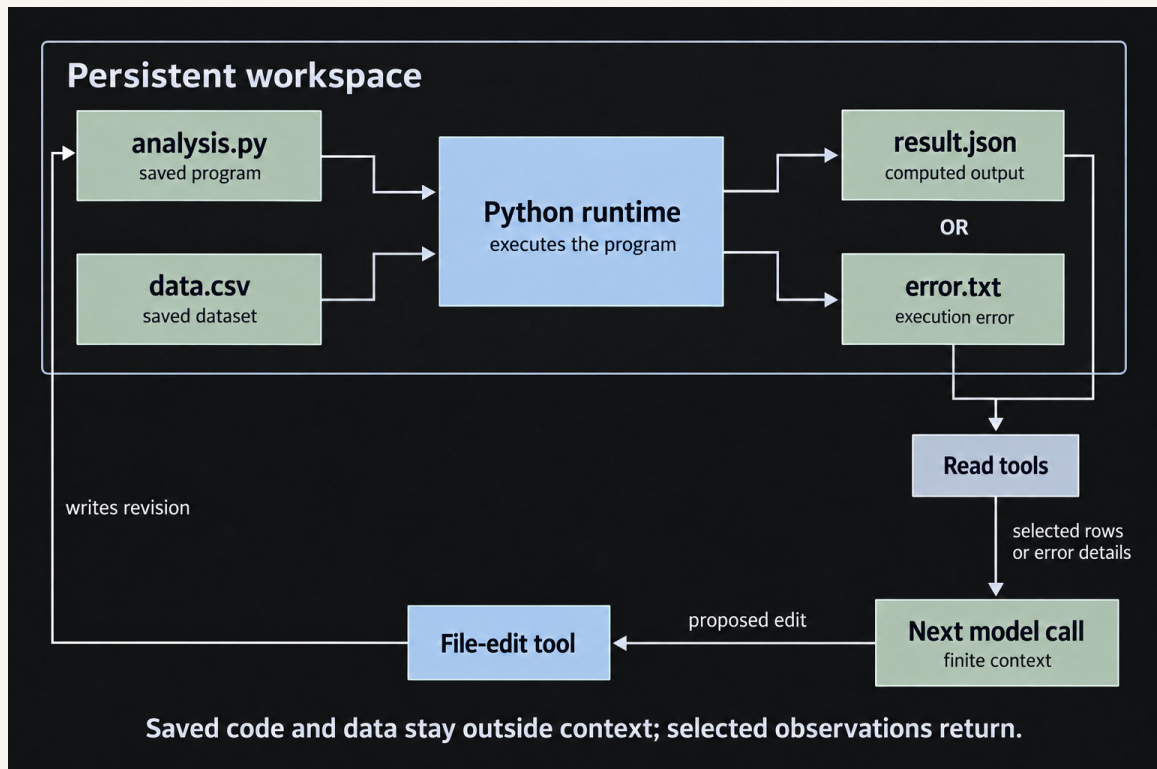
An artifact can also be input to a computation. A source file can be compiled or executed; a spreadsheet can recalculate formulas; a dataset can support a query. The file or object has meaning to the software that operates on it. By creating and changing these artifacts, an agent can arrange computations whose intermediate steps never need to pass through a model's context.

An agent can analyze a saved dataset by writing a program. The model produces a program, a tool writes it into the workspace, and a runtime executes it against the data. The runtime returns a result or an error that the agent can inspect. The following application interfaces save that outcome before selecting an observation for the next call. A large result can remain in storage while the model receives the rows or error details relevant to its next decision.


```
// Illustrative workspace and execution interfaces.
await workspace.write("analysis.py", program)
const outcome = await runtime.execute({
  program: "analysis.py",
  input: "data.csv",
})

const saved = await workspace.saveOutcome(outcome)
return {
  status: outcome.status,
  artifact: saved.path,
  observation: selectObservation(outcome, question),
}
```

The same arrangement supports iterative work. An error can guide an edit to the saved program; a query result can motivate another query; a rendered page can reveal a layout problem. Programs, data, and outputs accumulate in the workspace while each model call receives a selected view. The environment performs the execution, calculation, or rendering, and the agent interprets the result to choose what to do next.



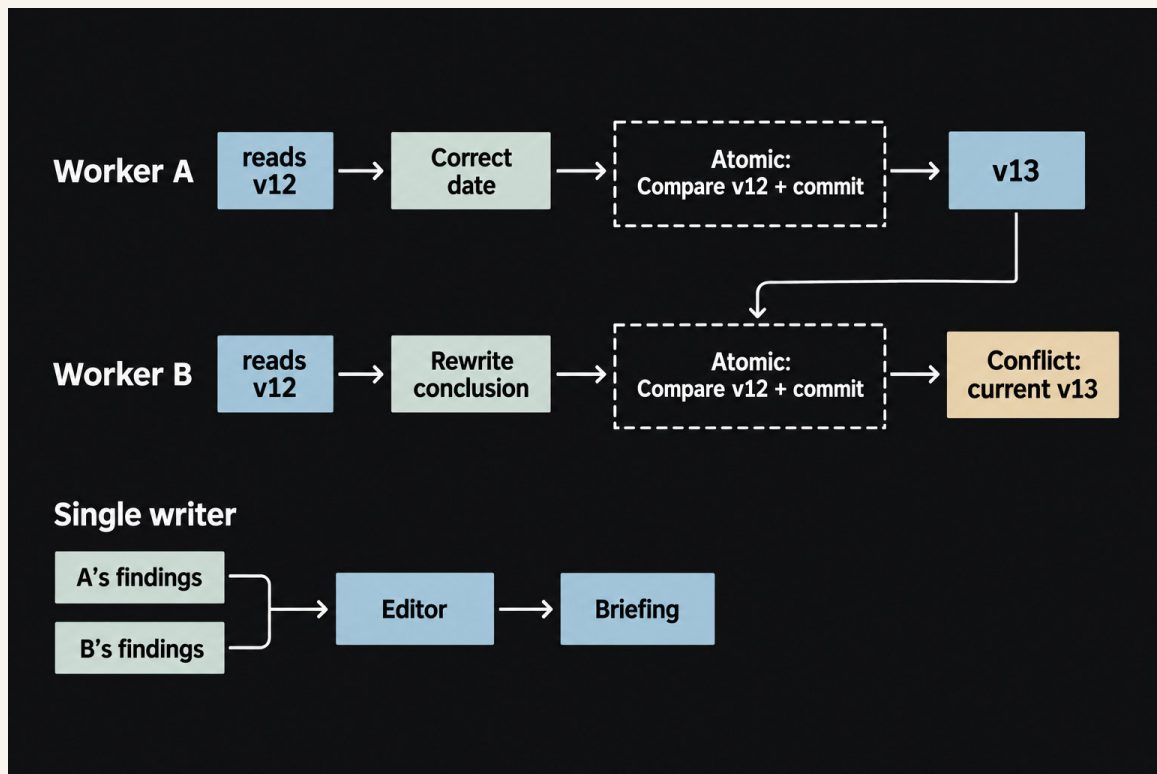
A runtime operates on saved code and data. Selected output or error details enter the next model context, where they can guide another tool-mediated edit.

Concurrent edits

Several agents can follow a protocol and still interfere with each other's work. Two writers may read revision 12 of a document, make different corrections, and each save a replacement. The later write can erase the earlier correction. A written instruction to respect ownership helps participants cooperate, but the storage operations determine whether conflicting updates can actually occur.

A single writer can integrate contributions from separate result files or proposed patches. Multiple writers can instead use version-checked updates: accept a change only if the revision it was based on is still current. The comparison and write must happen atomically, just as the task claim checks availability and records ownership together. Checking a revision and saving in separate unprotected steps leaves room for another writer to intervene.

A conflict gives the agent new information. The agent can read the current version, inspect the other contribution, and decide how to revise its proposal. Independent comments may both survive, while contradictory changes to a conclusion require a substantive decision. The environment can detect a stale update and preserve earlier work; the model or another participant still has to resolve what the combined result should mean.

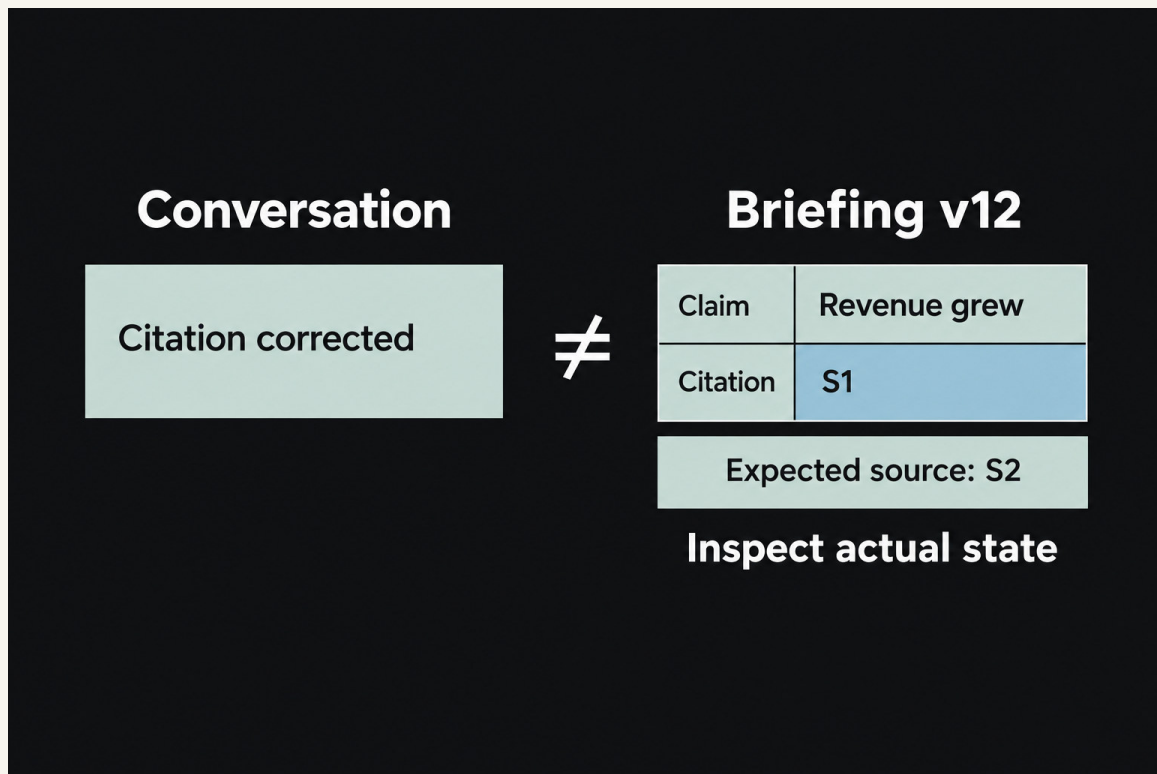


One writer can serialize changes. Multiple writers need an atomic version check and commit so a stale save cannot erase newer work.

Continuity across runs

Persistent work lets a later run continue without depending on the previous worker remaining active. The new run reads the task, its protocol, and the relevant artifacts to establish what has been saved and what remains to be done. A conversation that says a citation was corrected is an account of an action; the current document shows whether the correction is present. If an operation's outcome is unknown, the next run may need to check what actually changed before trying again.

The same object can serve several purposes over time. A document is a work product when edited, evidence when reviewed, and memory when retrieved for a later task. Its identity and history connect those uses. Saved content, task state, and the protocol together give a new participant a basis for continuing the work with a fresh context.



An operation changes external work. Inspecting the artifact establishes what actually changed, independently of what the model reported.

Artifacts let work and its procedures outlive individual agent runs. Their interfaces make contributions discoverable, provide operations for changing shared state, and connect the system to external computation. A stored task still needs a running participant to act on it. The next chapter asks what starts those later runs and how they resume a continuing purpose.

Autonomy

How agents start and resume work without another user message.



Events and scheduled messages give a system new occasions to resume its work.

An autonomous agent can keep working without a person sending every prompt. Something still has to start the next model call. A timer, an incoming message, or a changed file can do that. The agent also needs a record of what it was doing, so the next call can pick up the same work.

The execution cycle

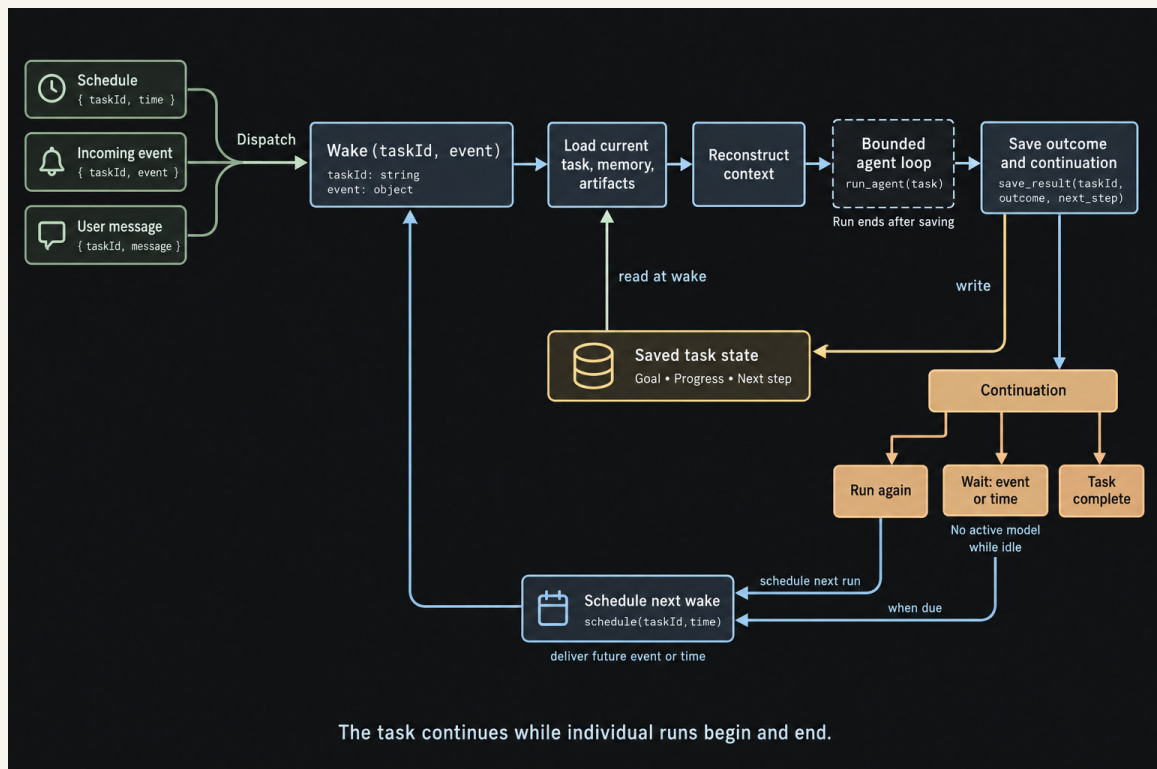
An agent can stop running while its task remains unfinished. It might be waiting for a reply, a new document, or tomorrow's data. A program can start the agent again when that information arrives, using the saved task and recent work as context. The agent loop handles the work within each run; this extra code connects one run to the next.

The agent needs to save enough information for the next run to continue: what it finished, what remains, and what it is waiting for. The wake helper below loads the saved task and gives it to the agent along with the event that woke it. The saved result includes what to do next. A timer or event handler can use that result to arrange another run.

```
// These helpers stand in for your application's code.
async function wake({ taskId, event }) {
  const task = await tasks.load(taskId)
  if (task.status === "complete") return

  const context = await reconstructContext({ task, event })
  const result = await runAgent(context)

  await tasks.recordRun({ taskId, event, result })
}
// The saved result says whether to run again,
// wait for a time or event, or finish the task.
```



A trigger starts a finite run. Saved progress and continuation connect that run to future execution.

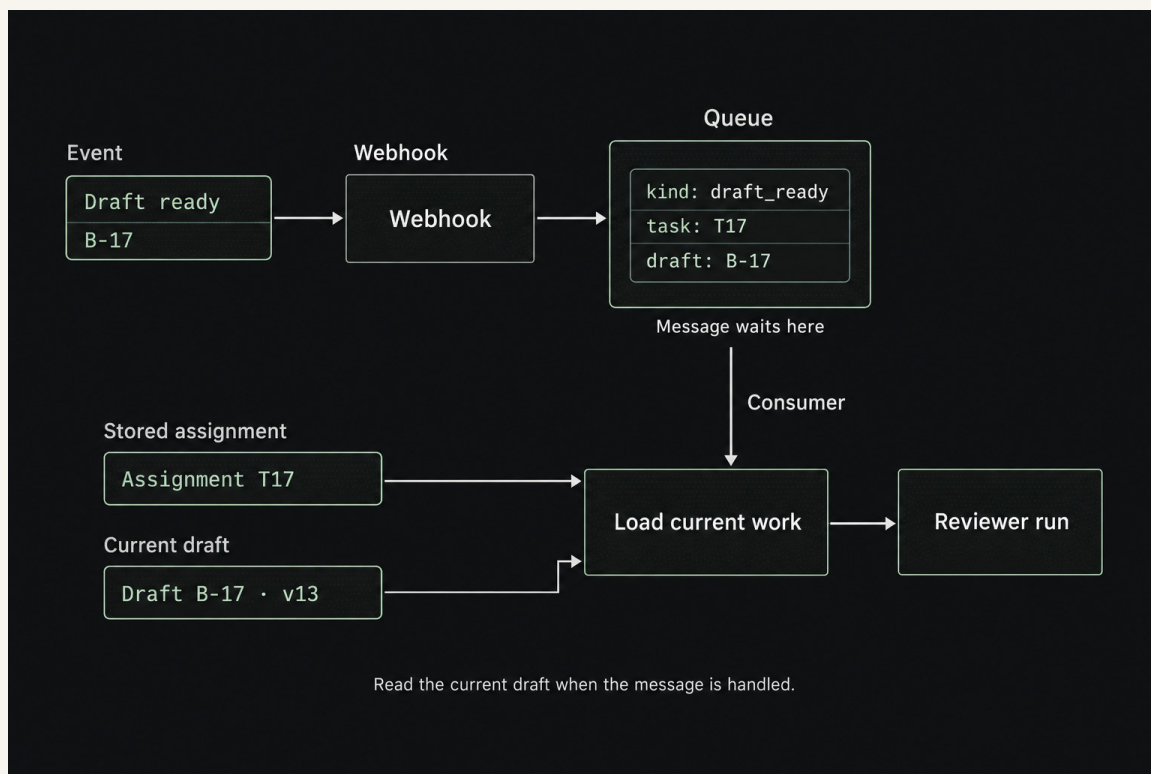
Triggers

An agent can wake when something changes. A draft marked ready for review gives a reviewer work to do. Starting that reviewer after every file edit would also wake it for changes unrelated to its task. Choosing the trigger determines which changes get the agent's attention.

Applications can receive or discover changes in several common ways:

- **Webhooks.** Another service sends your application a message when something happens.
- **File watchers.** Your application watches selected files or folders and receives notifications when they change.
- **Polling.** Your application checks a service regularly for changes. This is useful when the service cannot send updates itself.

If the agent cannot handle an event immediately, a queue can hold the message until it is ready. A message such as `{ taskId, kind: "draft_ready", draftId }` identifies the work. The program uses `taskId` to load the saved assignment and gives the agent `draftId` so it can read the draft. Reading the current draft matters because more edits may have happened while the message waited in the queue.



The webhook reports that a draft is ready. A queue holds the message until a consumer can load the assignment and current draft for the reviewer.

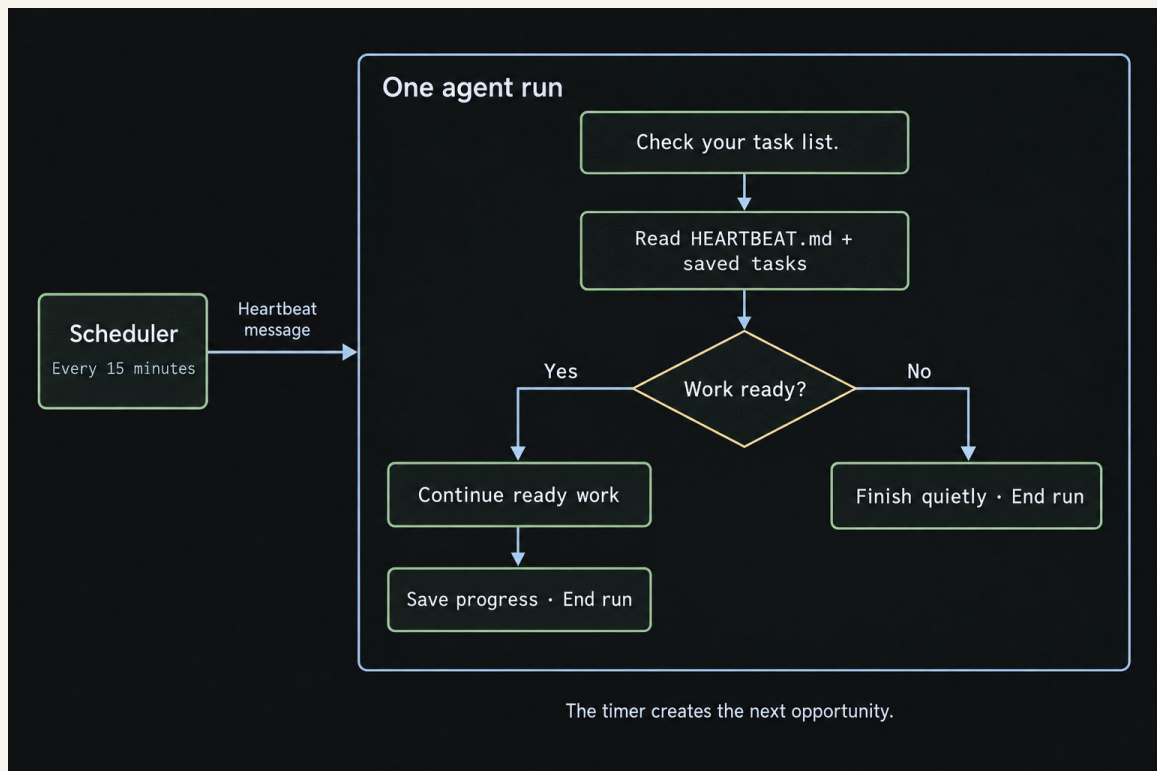
The Heartbeat Pattern

A heartbeat can be as simple as a timer that sends the agent a message every 15 minutes: “Check whether any new sources change the briefing.” That automated message starts an agent run, just as a user message would. The agent can use its usual tools to check for updates and do the work.

The message can also leave the choice of work to the agent: “Check your task list and continue anything that is ready.” A file such as HEARTBEAT.md can hold standing instructions about what to check. The agent reads those instructions and its saved tasks, then decides what needs attention. If nothing needs doing, it can finish quietly.

```
// The scheduler stores this repeating job and calls wake.
await scheduler.every("15 minutes", {
  handler: "wake",
  payload: {
    taskId,
    event: {
      kind: "heartbeat",
      message: "Check your task list and continue anything that is ready.",
    },
  },
},
})
```

The timer keeps time; the model does not have to keep thinking between checks. Shorter intervals catch changes sooner but cost more if they keep finding nothing new. Events can wake the agent for known changes, while a heartbeat gives it a regular chance to notice unfinished work. The agent does not need to send the user a message every time it checks.



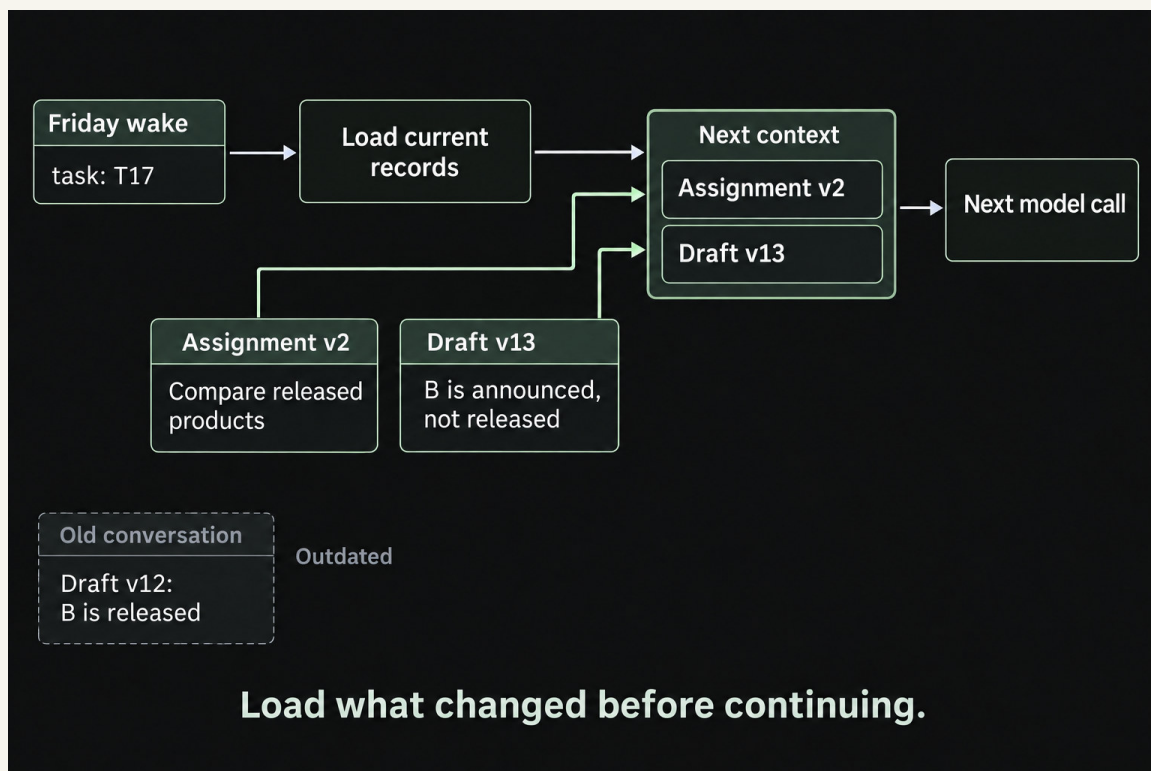
The timer starts a run every 15 minutes. Inside that run, the agent reads its instructions and tasks, then continues ready work or finishes quietly.

Context reconstruction

“Check your tasks” is only useful if the agent can find those tasks. The next model call needs the goal, relevant notes, and access to the current work. The program can include a short summary in context and let the agent read more from files or databases. This is the same memory problem from earlier chapters.

The last conversation may no longer describe the current work. Another agent may have edited the draft, or the user may have changed the assignment in a different thread. Replaying the old conversation alone can leave the agent working from outdated information. Saved history explains earlier decisions; current files and task records show where the work stands now.

Commitments need records too. If the agent promises to check again on Friday, the system needs to save both the reminder and a scheduled wake-up. Friday's model call can then recover what to check and why. Those saved records let the agent remember the commitment and follow through.



The wake identifies the task. Loading its revised assignment and current draft lets the next run continue from the latest work, even when the old conversation is outdated.

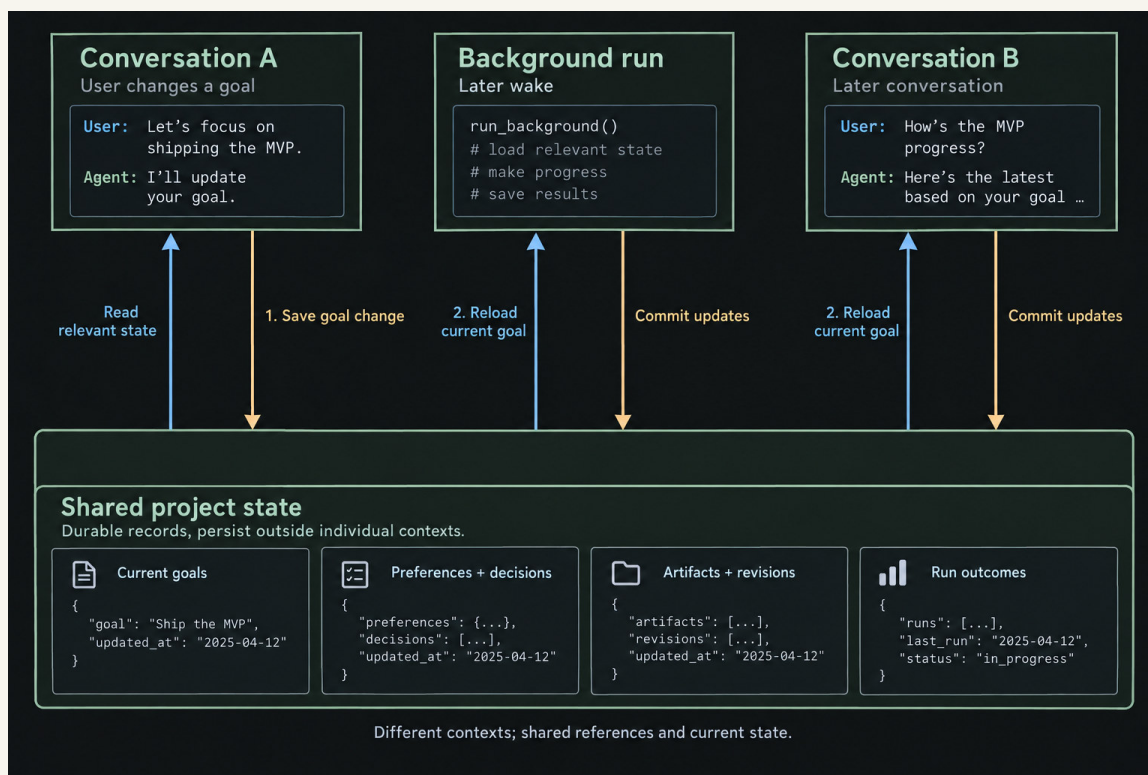
Continuity across threads

Two conversations with the same agent can have different histories. Background jobs may have their own histories too. If a decision stays inside one conversation, other runs will not know about it. Shared notes and task records let separate conversations and background work stay connected.

If the user changes a research question in chat, the agent should update the saved assignment that background jobs read. If a background job finds conflicting evidence, it should save the finding where a later chat can retrieve it.

Information needs to travel both ways for the system to act like one agent that knows what it has been doing.

Each call still needs only the information relevant to its work. Task details can stay with the task, lasting preferences in a user profile, and findings beside their sources. Shared task and project identifiers help each run find the right records. When several agents edit the same record, the version checks from Artifacts help prevent one update from overwriting another.



Separate contexts can continue shared work by saving changes and retrieving current goals, decisions, and artifacts.

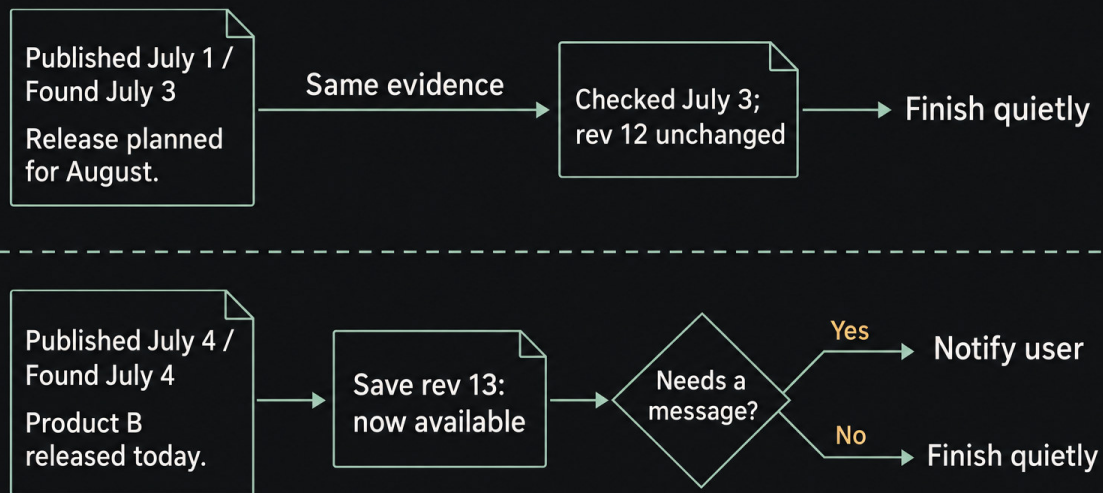
Follow-ups

A check can finish while the larger task continues. An agent monitoring sources may find nothing new today and still need to check tomorrow. It can also choose a useful time to return, such as after an expected release. A scheduling tool can save that request and arrange the wake-up. Writing “I will check tomorrow” in a response does not set a timer.

New to the agent does not always mean new to the task. A source found today may describe last week's events or repeat evidence already in the briefing. Recording when a source was published and when it was checked helps the agent tell a new finding from a real change. An uneventful check can simply record that the work is still current.

Saving work and notifying the user are separate choices. The agent can update a note quietly and send a message when a finding changes the conclusion or needs a decision. The next conversation should be able to recover the saved result either way. Frequent checks can be useful without becoming frequent interruptions.

Briefing rev 12: release unconfirmed



A newly found source may repeat existing evidence. The agent can record an unchanged check quietly; when a source changes the briefing, it saves the revision before deciding whether to notify the user.

A timer can keep an agent checking, and shared records can help it pick up where it left off. We now have a system that can keep working between conversations. The next question is how to tell whether that work is any good.

Evaluation

How to establish whether the work succeeded.



Evaluation compares the work with a requirement using evidence that can be inspected.

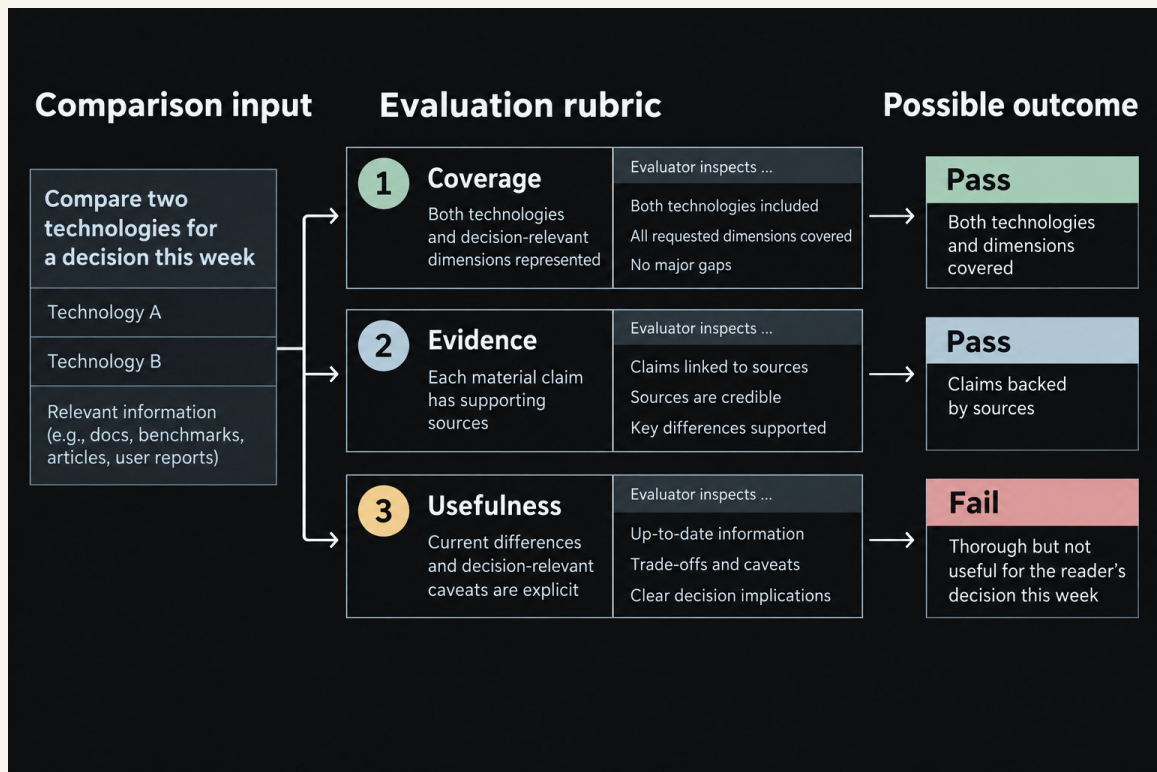
Evaluation is how we judge whether an agent's work meets the task's requirements. The systems built so far can keep working across many model calls, but finishing a run does not establish that the task succeeded. A convincing answer may contain a mistake that the agent never noticed. To check the work, we need to decide what counts as success, obtain evidence about the result, and compare the two.

Criteria

A criterion is a condition we check to judge whether the agent did its job. It connects the task we gave the agent to a result we can inspect:

- **A research agent's answer:** the cited sources support the factual claims in the answer.
- **A coding agent's bug fix:** a test that reproduces the reported bug passes against the edited code, and existing tests still pass.
- **An agent's document update:** the saved document contains the requested change.

A task may need several criteria. The research answer may need to cover every part of the question as well as support its claims. A set of criteria is often called a rubric. The rubric tells the evaluator what to check while allowing different answers or approaches to satisfy the task. Keeping the results for each criterion makes it clear where the work fell short, which an overall score alone cannot explain.



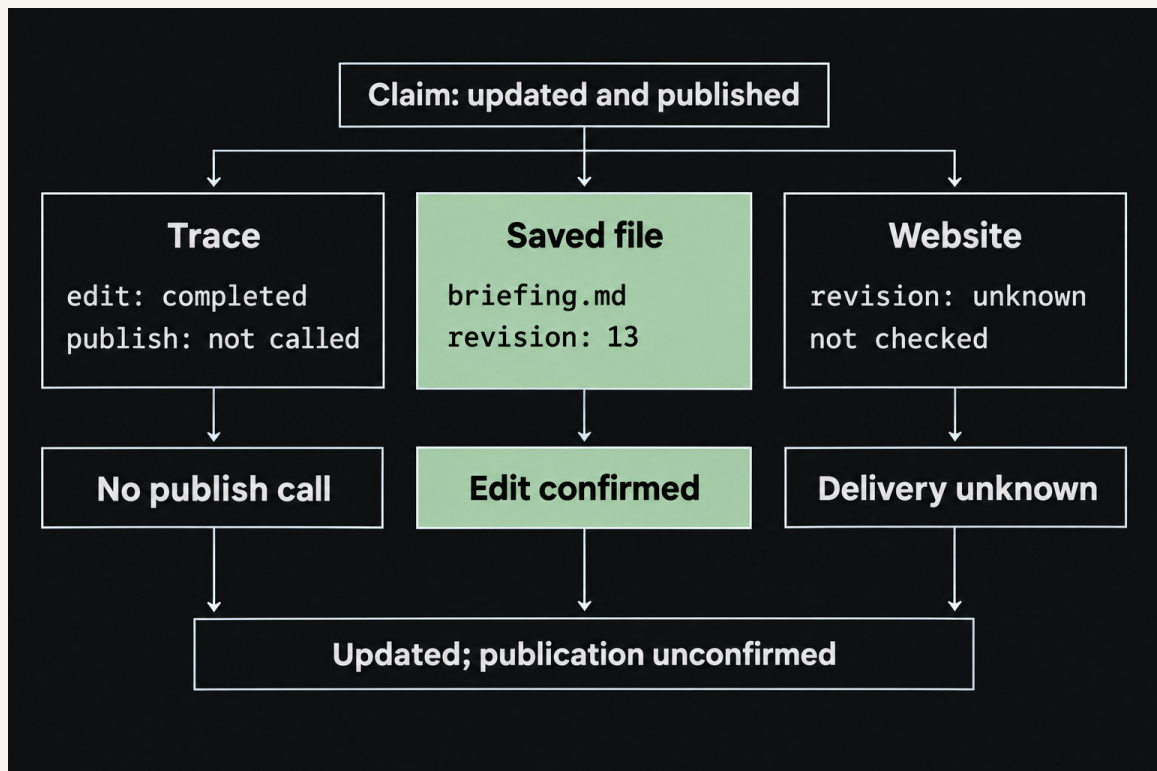
Quality becomes testable when requirements name what must be observed, rather than merely asking for a good answer.

Evidence

Once we know what to check, we need the material that can answer the question. To assess a factual claim in the research answer, the evaluator needs the claim and its supporting source. The agent's assurance that it checked the source adds little if we cannot inspect what the source says. This connects evaluation to the artifacts from earlier chapters: saved work gives a reviewer something to examine after the run ends.

Different requirements call for different evidence:

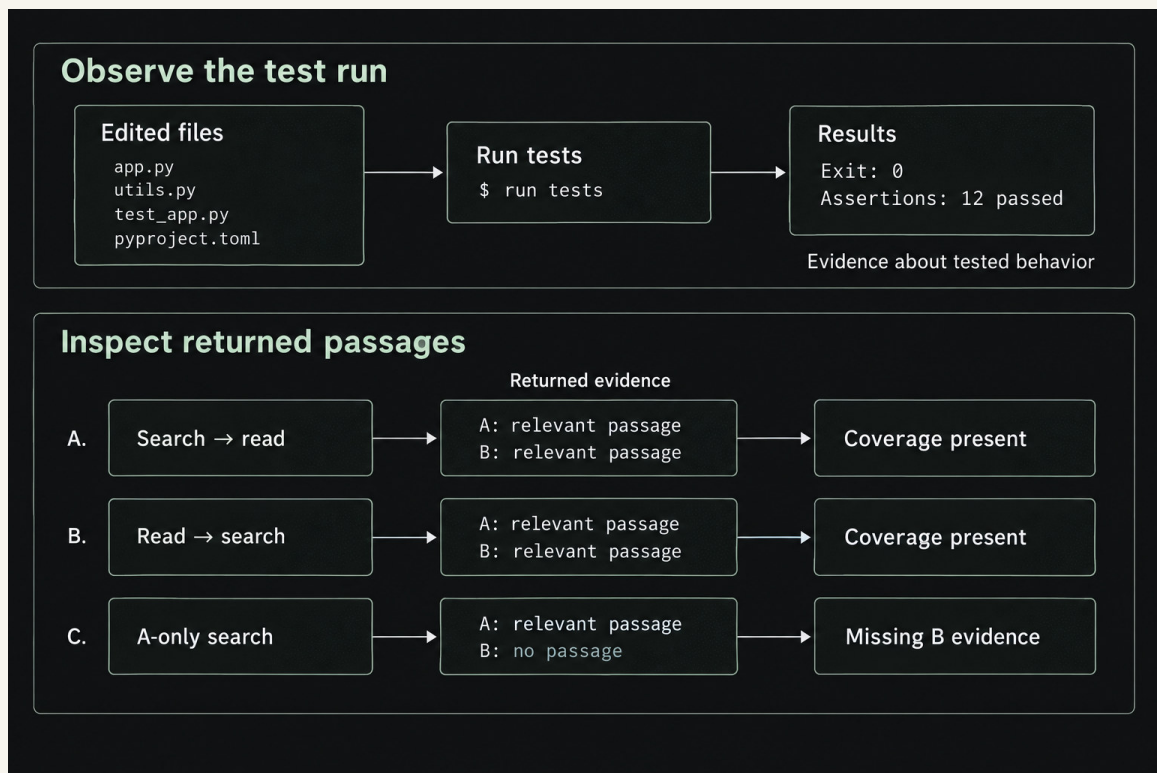
- **The final answer:** the claims, explanations, and formatting the reader receives.
- **The resulting state:** the actual file, document, or database record after the agent's action.
- **The execution trace:** the recorded operations, their results, and the time or resources they consumed.



The saved briefing confirms an edit. The trace records no publish call, and the destination has not been checked, so publication remains unconfirmed.

If an agent says it fixed a bug, we can run tests against the changed files and record the results. Asking the model whether those tests would pass gives us its prediction. Running them gives us observations about the code.

The trace helps when the requirement concerns how the work was done. If a comparison must examine both company filings and product documentation, the recorded results can show whether the agent received relevant passages from both collections. A search call alone does not establish that coverage: we need to inspect what it returned. Several orders of searching and reading can satisfy the requirement, so the check should allow those alternatives unless the order itself matters.



Running tests produces observations about the edited files. Research coverage likewise depends on the relevant passages returned, even when the operations occur in a different order.

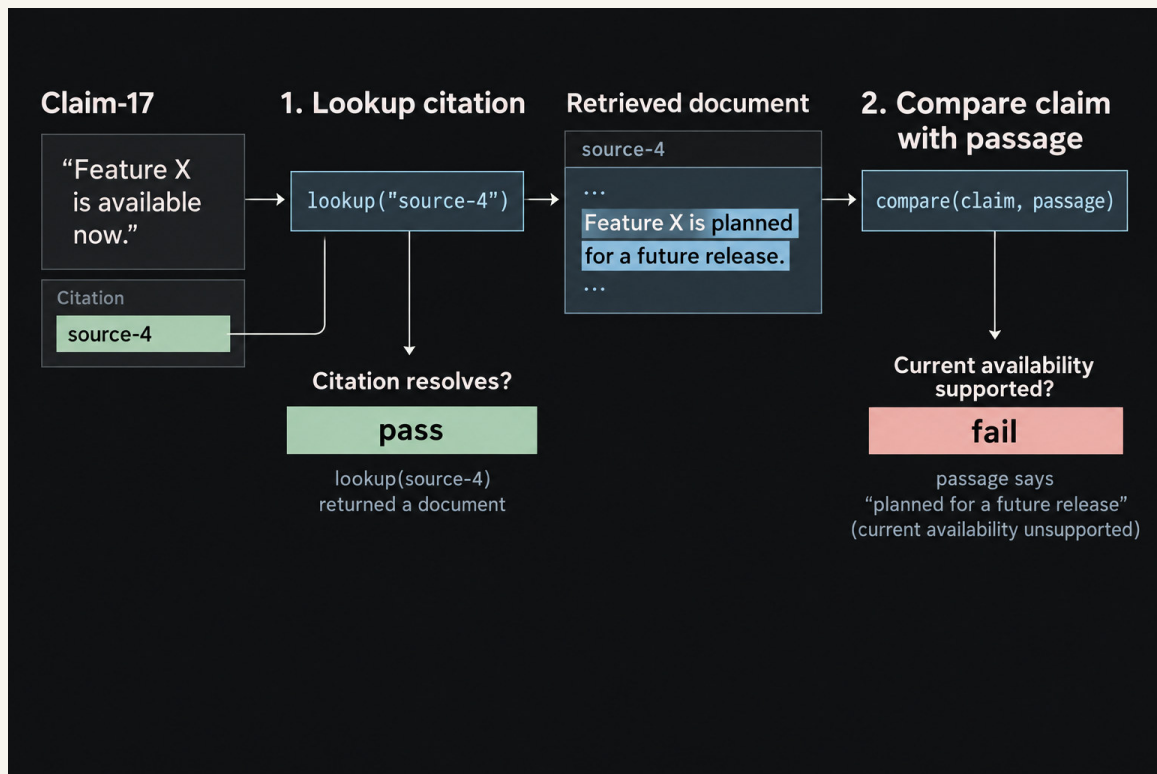
Tests and validators

When a requirement has a precise rule, ordinary code can check it. A validator is a function that checks whether data follows such a rule. For citations, we can look up each source identifier in the stored sources. The example below receives a claim with a source identifier and a map of available sources. Its result names the condition checked and the claim it applies to.

```
function checkCitation(claim, sources) {
  const exists = sources.has(claim.sourceId)

  return {
    criterion: "Citation points to a stored source",
    result: exists ? "pass" : "fail",
    location: claim.id,
    evidence: exists ? [claim.sourceId] : [],
  }
}
```

This check can pass while the claim is wrong. A research answer written on July 1 might say “Product B is available now,” while a source published that day says “Product B is scheduled for release next month.” The source describes a planned August release, so it does not support availability on July 1. The lookup correctly establishes that the source exists. Checking whether it supports the claim requires reading what the passage means. A check can work exactly as intended while leaving an important requirement untested.



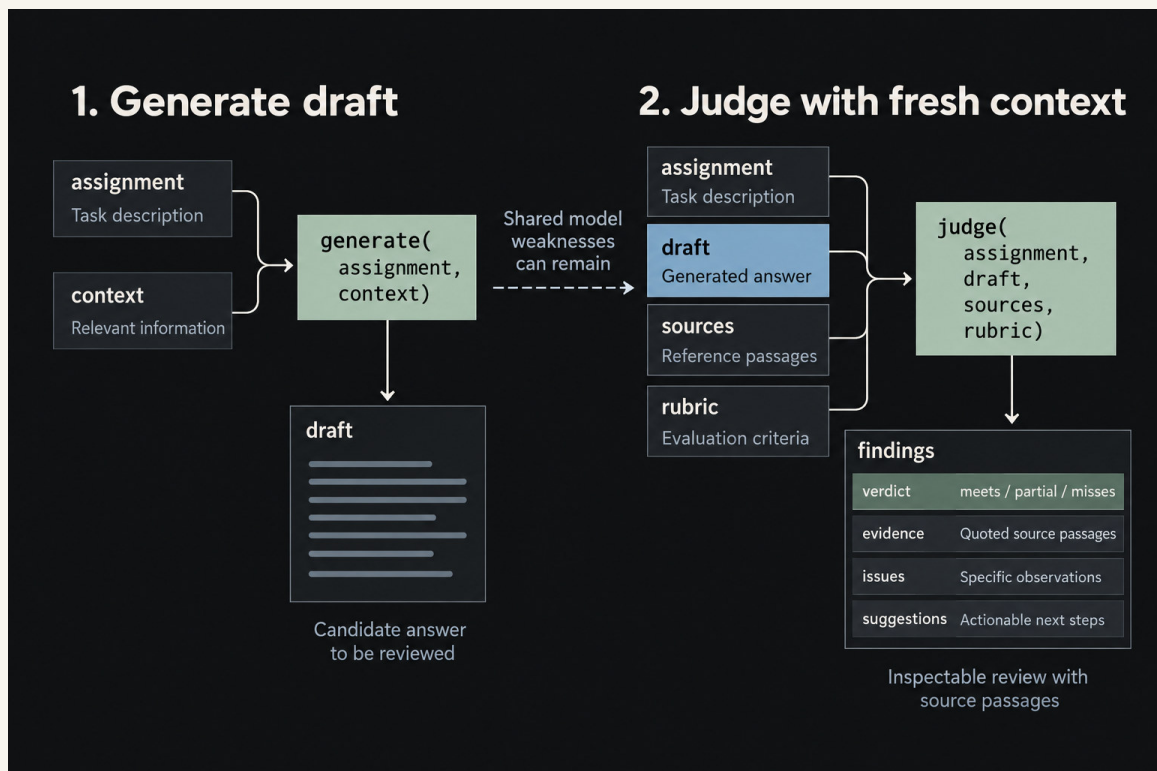
A citation lookup can establish that a source exists. Source support requires a further check.

Model-based review

A language model can help evaluate requirements that depend on meaning. In our citation example, the model can read the source passage and assess whether it supports the claim. This is often called using an LLM as a judge. The reviewer needs context just as the writer did: the task, the work being judged, the criteria, and the evidence relevant to those criteria.

A separate review call lets us select that context for the check. The reviewer need not receive every search attempt or the writer's explanation of why its draft is good. In the example below, the application supplies the actual claim and source passage, then asks for a judgment it can inspect. The `callModel` helper represents the same kind of model adapter used in earlier chapters; it formats these instructions and inputs into a request.

```
const review = await callModel({
  instructions: [
    "Does the source support the claim of current availability?",
    "Return pass, fail, or unknown.",
    "Include the claim ID, your explanation, and quoted evidence.",
    "If evidence is unavailable, return unknown.",
  ].join("\n"),
  input: {
    assignment: "Compare products available to use this week.",
    assessmentDate: "2026-07-01",
    claim: {
      id: "claim-17",
      text: "Product B is available now.",
      sourceId: "source-4",
    },
    source: {
      id: "source-4",
      publishedAt: "2026-07-01",
      text: "Product B is scheduled for release next month.",
    },
  },
},
})
```

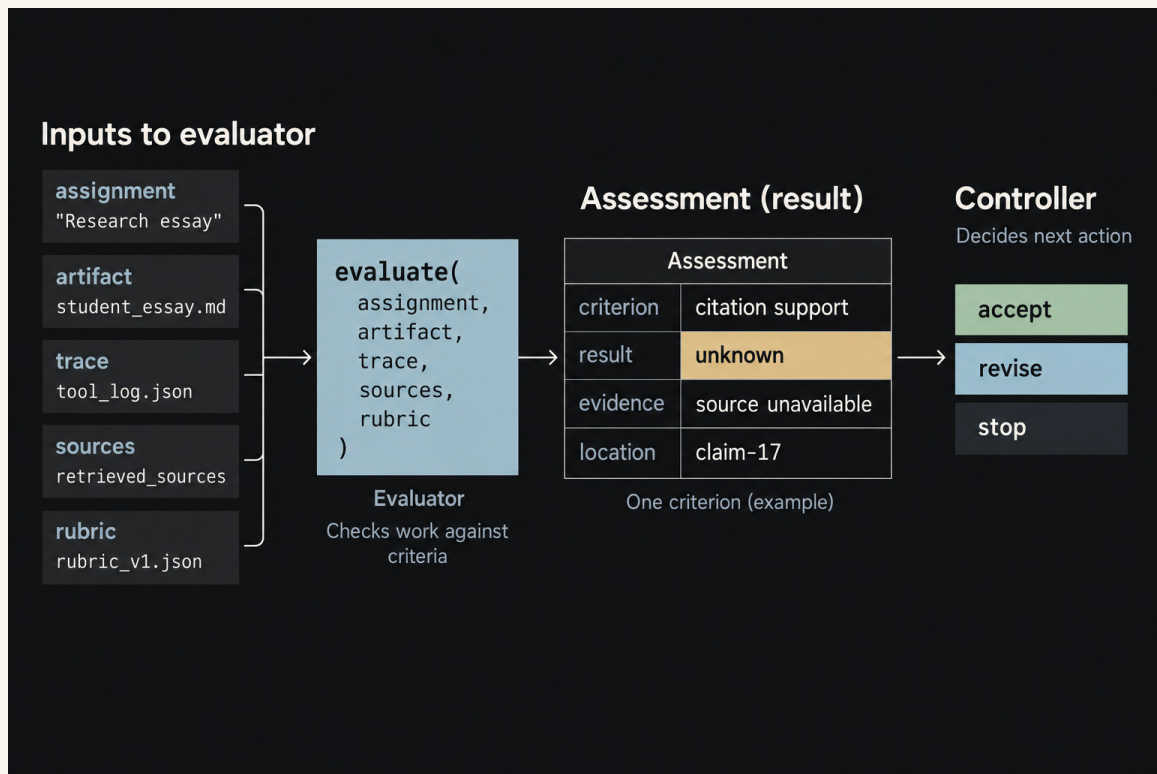


A fresh reviewing context can focus on the assignment and evidence. It still needs calibration and can share the generator's errors.

A useful finding identifies the problem and the evidence behind it. For this example, the reviewer should report that the source describes a future release and therefore does not support the claim of current availability. The application can represent that finding as a record. Parsing and validating a structured response checks its format; the quality of the judgment still depends on the review.

```
// An illustrative finding from the review above.  
const finding = {  
  criterion: "Source supports current availability",  
  result: "fail",  
  location: "claim-17",  
  evidence: [  
    "source-4: Product B is scheduled for release next month.",  
  ],  
  explanation: "The source describes a future release.",  
}
```

Code checks and model reviews can both produce records in this form. Each record says what was checked, the result, and the supporting evidence. An unknown result preserves the difference between a failed requirement and a requirement we could not assess. A score can help compare drafts; a finding that identifies the faulty claim can help correct it. The information an assessment retains depends on the decision it will support.



An evaluator combines observations with criteria and returns findings. A controller decides how to use them.

Calibration

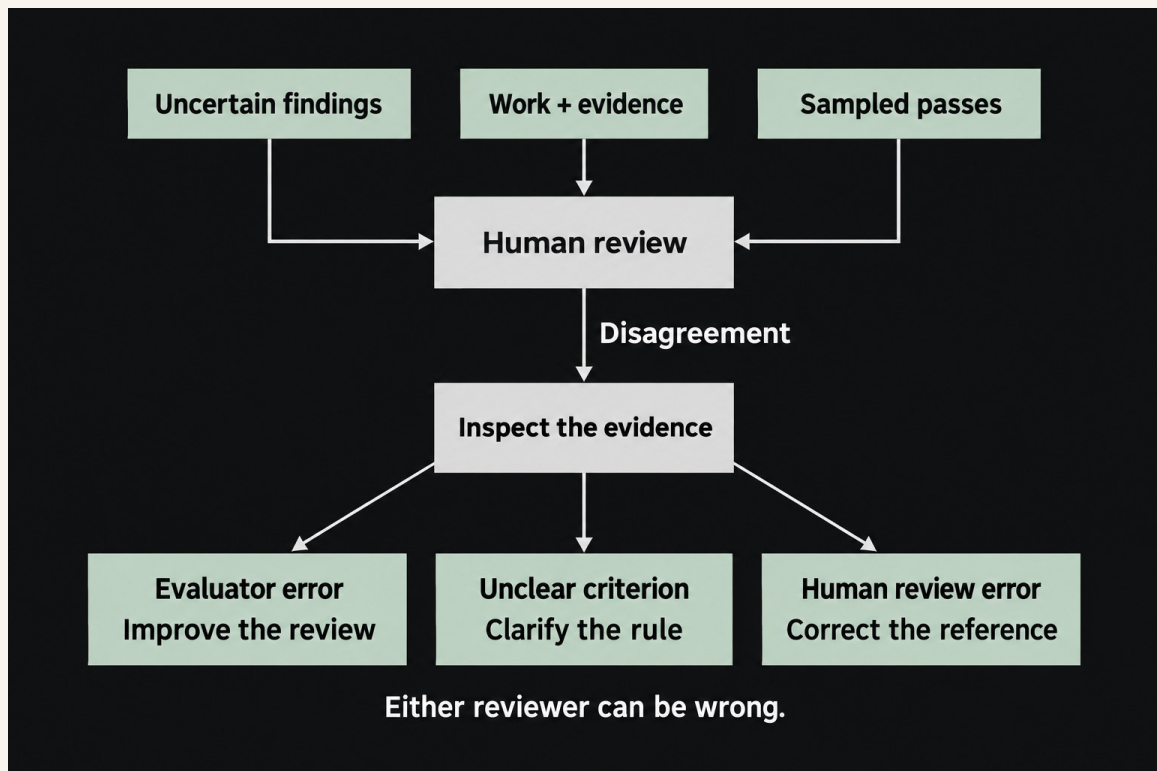
A reviewer can make mistakes too. A separate call gives the model a focused reviewing task, but it can still overlook the same ambiguity as the writer. It may reward confident prose or accept a citation without reading it closely. Self-review can help when it checks specific evidence, yet neither a fresh context nor an extra review guarantees a correct judgment.

Calibration means checking whether the evaluator's judgments match the standard we intended. We can have people review a small set of examples, compare their findings with the evaluator's, and examine disagreements. The reasons for those judgments help us decide what needs to change:

- **An unsupported claim passes:** check whether the evaluator received the relevant source and understood what to look for.
- **Reviewers disagree about a public beta:** clarify whether the task counts a beta as a released product.

- **A human reviewer misses evidence:** correct the reference judgment used to assess the evaluator.

Different checks can work together. A source lookup can catch a missing citation before a model reviews the claim's meaning. A person can examine uncertain findings and a sample of accepted work. Sampling accepted work matters because an evaluator's missed mistakes appear among its successes. Checking a test or validator follows the same principle: use examples whose expected results are known, including cases that should fail.



Review uncertain findings and a sample of apparent successes. When judgments disagree, inspect the evidence before deciding which judgment or criterion needs correction.

Reliability

Evaluating one result tells us about that attempt. To understand how reliably the system works, we need to run it on a range of tasks. Model responses can also vary when the task stays the same, so repeated attempts can reveal failures that one successful demonstration misses.

A small evaluation set can start with variations of the citation example:

- A release note clearly confirms that the product is available.
- An announcement describes a release planned for the future.
- Two sources disagree, or an older source has been superseded.
- The necessary source or publication date is missing.

Running the system on these cases produces a set of results and assessments we can compare. Repeating selected cases shows whether the same situation produces inconsistent answers. Grouping failures by cause shows where the system struggles: a high overall score can conceal repeated mistakes on future announcements. Timing and cost measurements help establish whether an improvement in quality is worth the additional work.

Illustrative cases

Same questions, different outcomes across trials.

Case	Trial A	Trial B	Trial C
Routine comparison	pass	pass	pass
Conflicting sources	pass	fail	correctly abstained
Missing date	fail	correctly abstained	pass
Obsolete source	fail	fail	correctly abstained
Insufficient evidence	correctly abstained	fail	fail

pass
Provided a correct answer

fail
Provided an incorrect answer

correctly abstained
Appropriately withheld an unsupported answer

Retained per attempt

Each attempt records the following fields to enable analysis across trials.

success

failure category

cost

elapsed time

severity

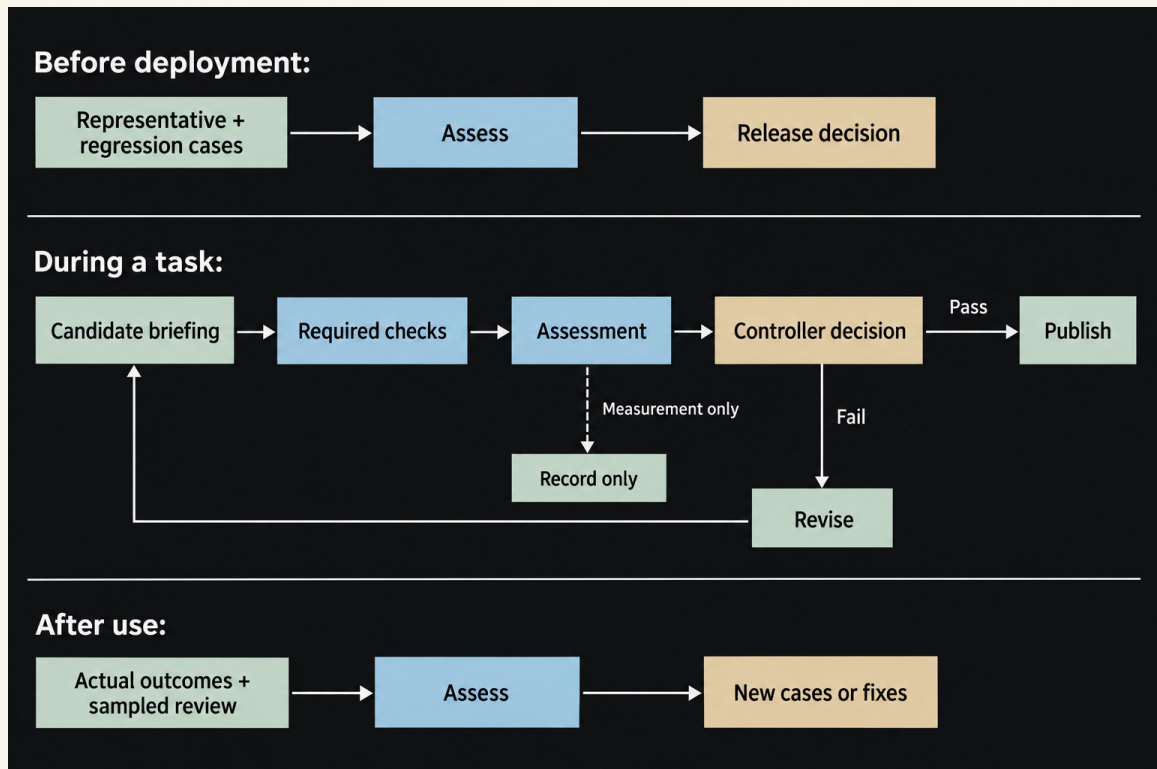
Repeated trials and failure categories reveal weaknesses that an average score can hide.

Evaluation can help at several points:

- **Before a change:** run saved cases to compare a proposed prompt or tool with the current system.
- **During a task:** check citations in a draft before returning it.

- **After use:** review actual outcomes to find failures the original test set never covered.

The right amount of evaluation depends on the work. A quick lookup may be worth running for every citation, while a detailed model review takes more time and a human review requires someone's attention. We can combine inexpensive checks with selective review, then use what we learn from real failures to improve the cases and criteria.



Before release, during a run, and after use are different opportunities to inspect and improve the system.

Evaluation gives the system a judgment about its work and, when possible, evidence explaining that judgment. Saving the finding does not correct the draft. The next chapter, Feedback, connects those findings to the next model call or action so the system can respond to what it learned.

Feedback

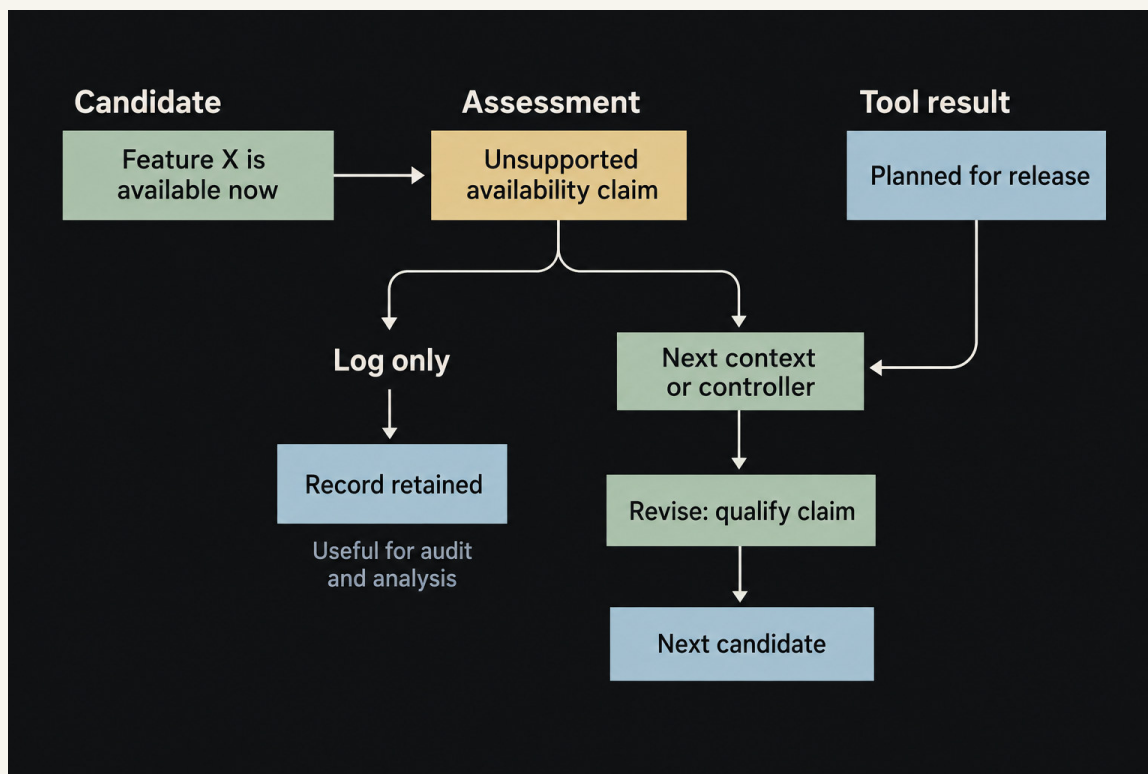
How observations change the next attempt.



An observation becomes feedback when it changes a later action.

Feedback uses the results of an agent's work to guide what it does next. Evaluation tells us where the work succeeds or fails. To improve the work, the system must use that assessment: give the findings to a model that can revise the answer, or let code choose the next action. An assessment saved only for reporting cannot correct the work by itself.

The agent loop already returns tool results to the model. But another turn does not necessarily bring the task closer to completion. The model needs information that helps it decide what to change, what to keep, or whether to try another approach. Designing feedback means making that information available where the next decision happens.



Feedback exists when an observation influences the next decision. A stored assessment alone leaves the task unchanged.

The Revision Loop

A revision loop gives the model a chance to correct its work after checking it. The program evaluates a draft, supplies the findings with that draft to another model call, and checks the revised version. Repeating these operations lets the system work toward the task's requirements instead of depending entirely on its first answer.

The citation example from Evaluation makes the connection concrete. The draft says a product is available now; its source describes a future release. The revision call receives the original task, the draft, the source, and the finding that identifies the unsupported claim. It can remove or qualify the claim. The next evaluation checks whether the revised answer now agrees with the evidence.

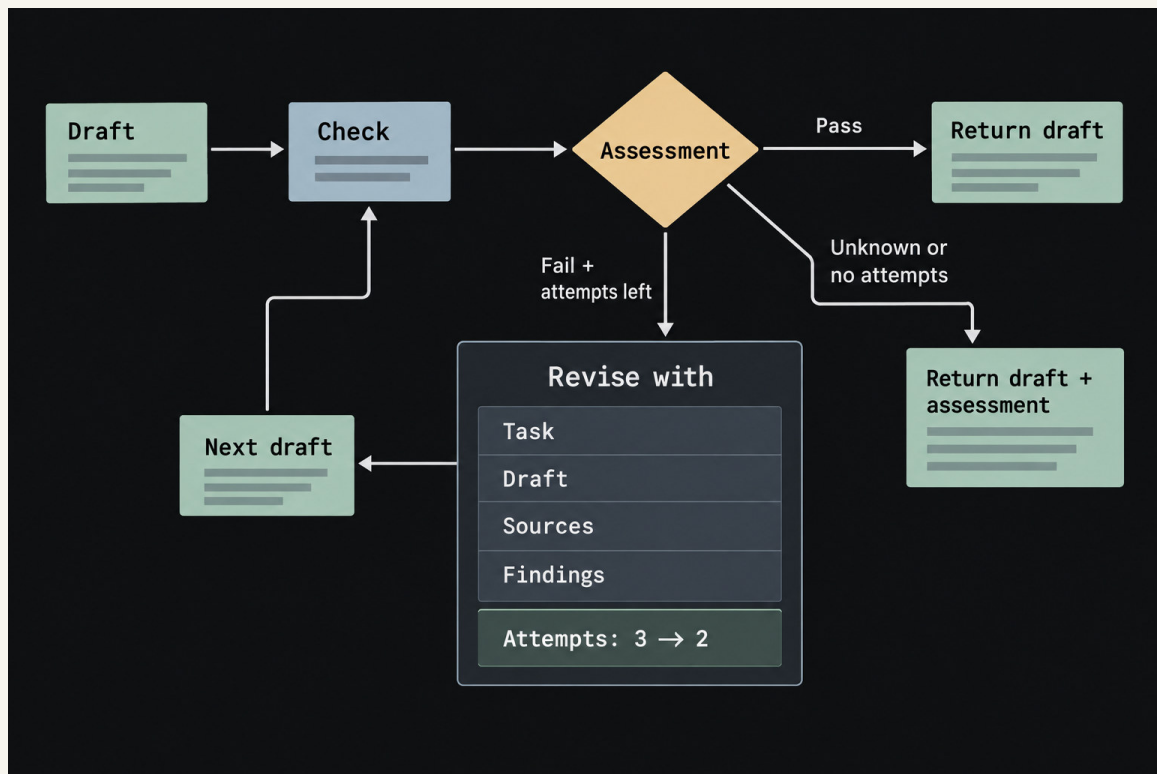
```
let draft = await callModel({
  instructions: "Answer the task using the supplied sources.",
  input: { task, sources },
})
let assessment = await evaluate({ task, draft, sources })

for (let attempt = 0; attempt < maxRevisions; attempt++) {
  if (assessment.result !== "fail") break

  draft = await callModel({
    instructions: "Revise the draft using the findings.",
    input: {
      task,
      sources,
      draft,
      findings: assessment.findings,
    },
  })
  assessment = await evaluate({ task, draft, sources })
}

return { draft, assessment }
```

Here, `evaluate` applies the checks from the previous chapter and returns `pass`, `fail`, or `unknown`, together with its findings. The loop retries failures up to `maxRevisions`. An `unknown` assessment returns to the application, since rewriting alone may not supply missing evidence. This simple loop returns the latest draft with its assessment; keeping earlier versions adds the option of backtracking, which we discuss below.

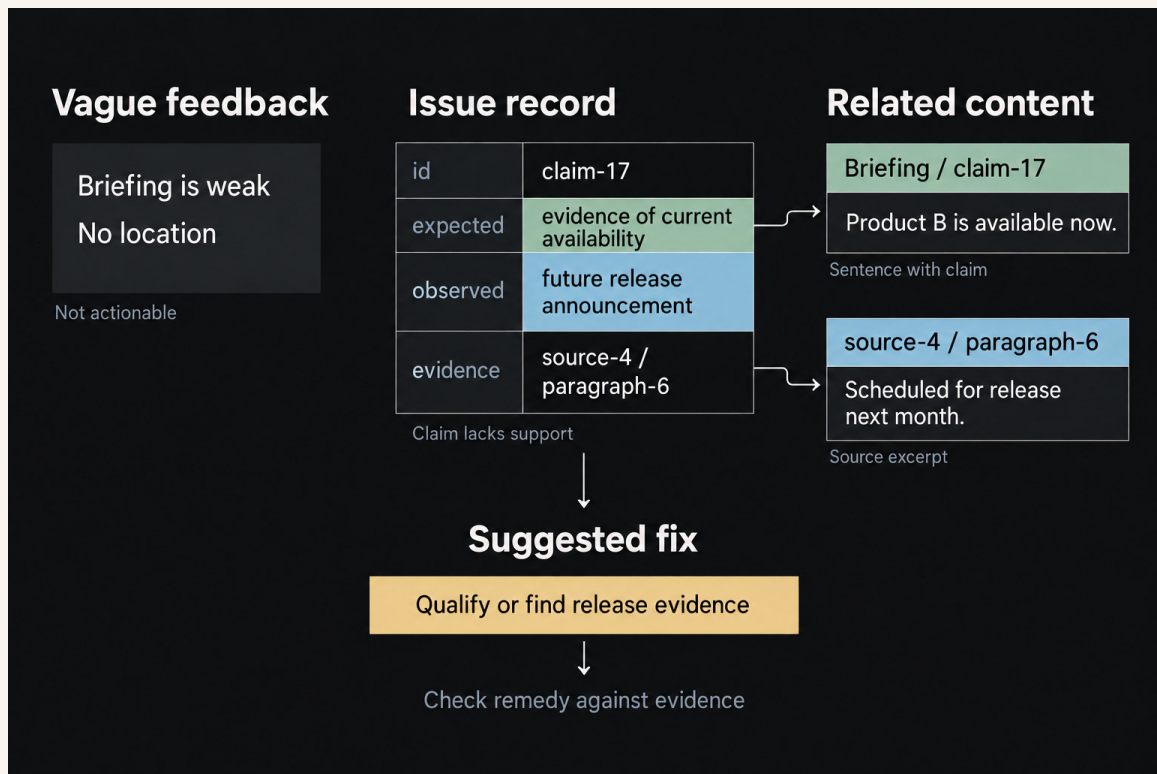


A failed check can trigger another revision with the task, draft, sources, and findings. Each revision consumes an attempt; a pass or an unresolved check ends this loop.

Diagnostic feedback

The usefulness of a revision depends on what the feedback tells the model. “The answer is wrong” leaves it to guess which part failed. A finding that identifies the claim, quotes the relevant source, and explains why it does not support the claim narrows the problem. The model can spend its next attempt addressing that discrepancy instead of rewriting unrelated parts.

A score can tell the program which draft to prefer or whether a threshold was met. To direct a repair, the model usually needs more detail. In the citation example, the useful information is why the source fails to support the claim. A suggested replacement can help, but it is a proposal to check against the source; the reviewer may have diagnosed the problem correctly while suggesting another unsupported claim.

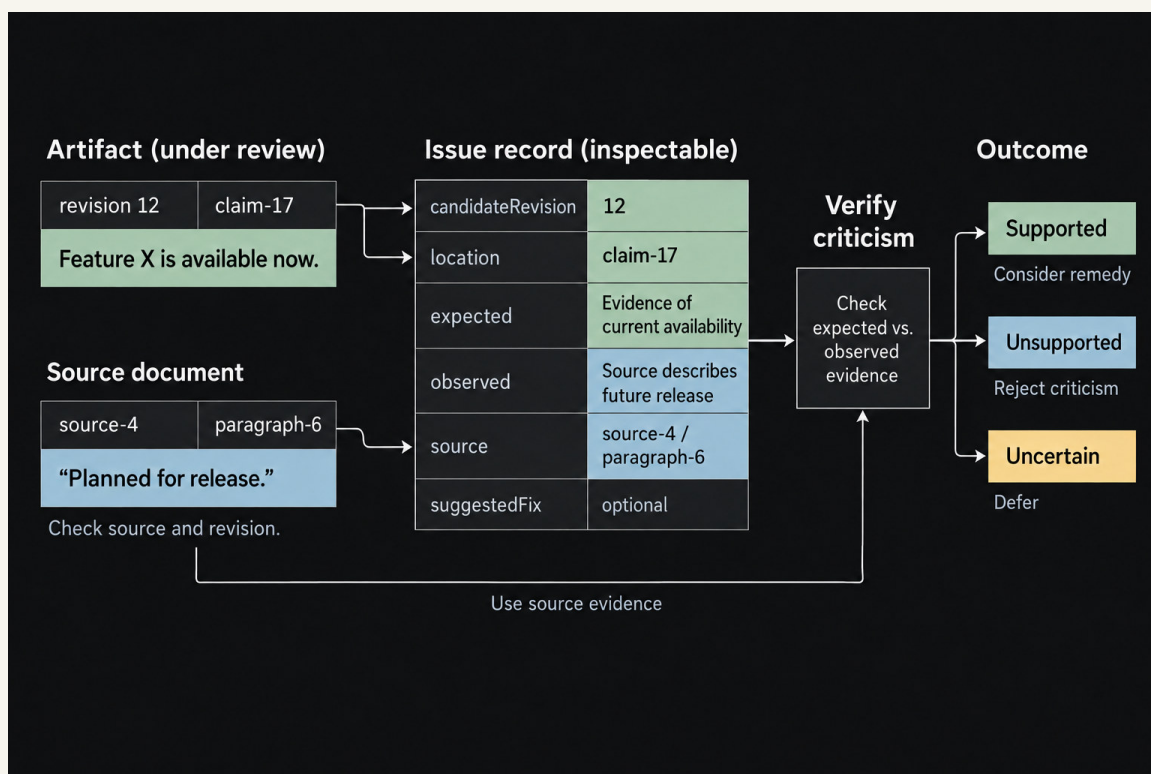


A finding identifies the unsupported claim and its source. The suggested fix remains a proposal to check against the evidence.

Coding agents have useful sources of this kind of feedback. A type checker can report that a function expected a number but received a string, with a file and line number. The agent can inspect that expression and its callers to decide what needs changing. Rerunning the checker then tests the change. The diagnostic narrows the investigation without having to prescribe the correct fix.

Other tasks need checks that provide comparable detail. For a research answer, the finding from Evaluation can travel with the source passage and the version of the draft it concerns. The passage lets the model inspect the evidence immediately. The version lets the application check whether the finding still applies to the current draft.


```
const feedback = {
  candidateRevision: 12,
  location: "claim-17",
  expected: "Evidence of current availability",
  observed: "The source describes a future release",
  source: {
    id: "source-4",
    passage: "Product B is scheduled for release next month.",
  },
  suggestedFix: "Qualify the claim or find release evidence",
}
```



A finding identifies the inspected revision and its source evidence. The application can check whether it still applies to the current draft.

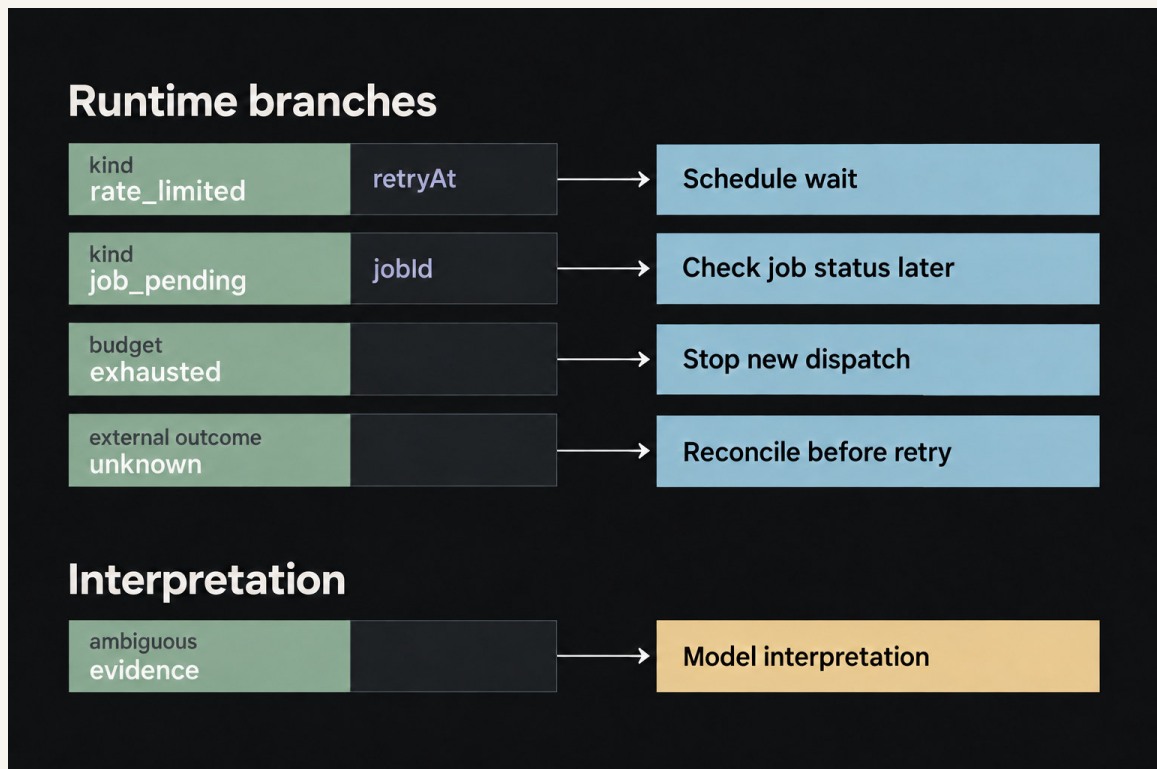
Responses to tool results

Not every problem needs another draft. The next step depends on what the result tells us about the work. An agent that keeps rewriting a request while a service is unavailable is changing the wrong thing. Tool results should preserve enough detail to distinguish problems the agent can fix from conditions it needs to wait for.

Common responses include:

- **Invalid tool arguments:** return the invalid field and the expected format so the model can correct its request.
- **A rate limit:** use the service's retry time to schedule another attempt.
- **An unfinished background job:** keep the job identifier and check its status later.
- **An uncertain save:** read the destination to find out whether the write succeeded before repeating it.

Code can handle a known response directly. A result such as `{ kind: "rate_limited", retryAt }` contains what the scheduler needs to arrange a retry. A result that requires interpretation can enter the next model call instead. If a search returns irrelevant sources, the model can examine what it found and change the query. Both paths use feedback; they differ in who chooses the response.

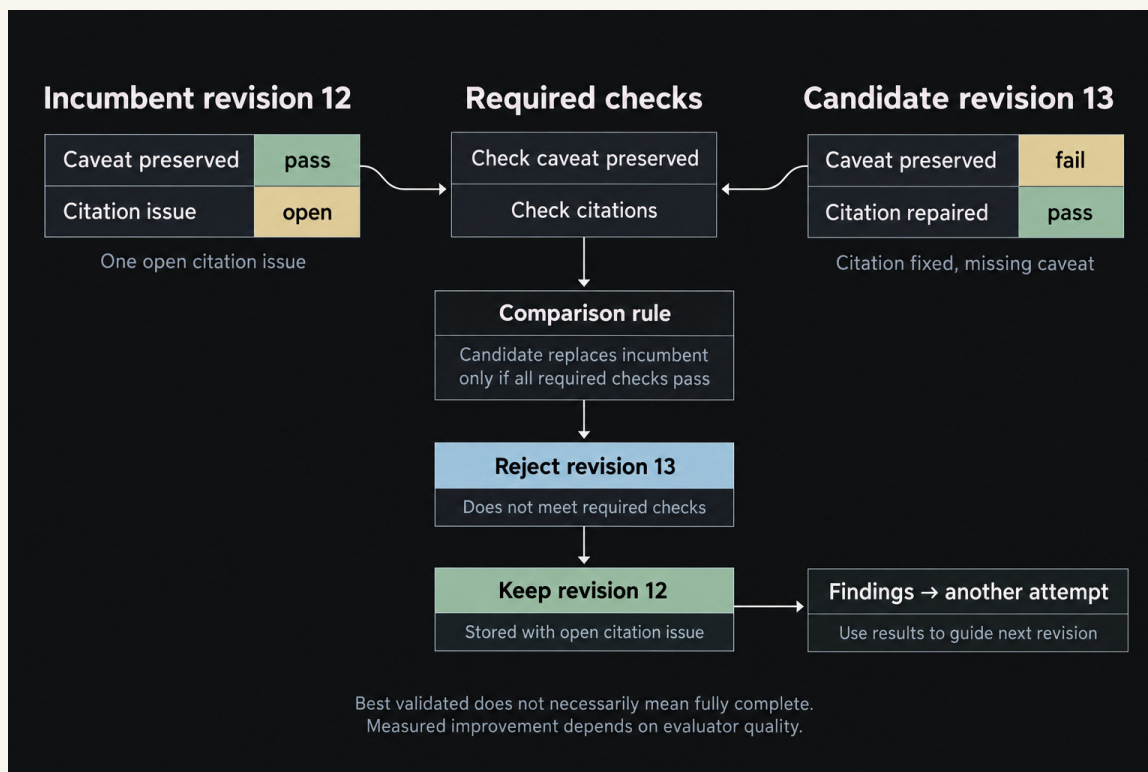


Known operational conditions can select a next step directly; ambiguous evidence may need model interpretation.

Backtracking

A revision can fix one problem and introduce another. Correcting an unsupported claim might also remove a caveat the answer needed. Checking only the reported defect would miss that loss. Regression checks revisit requirements that earlier work already satisfied, so a local repair does not silently undo progress.

Keeping a previous version gives the system a way back. The application can save the draft and its assessment, try a revision, and compare the results before replacing the saved version. If the revision loses required information, the next attempt can start from the earlier draft with that new finding. Versioned artifacts make this possible without asking the model to reconstruct the old text from memory.

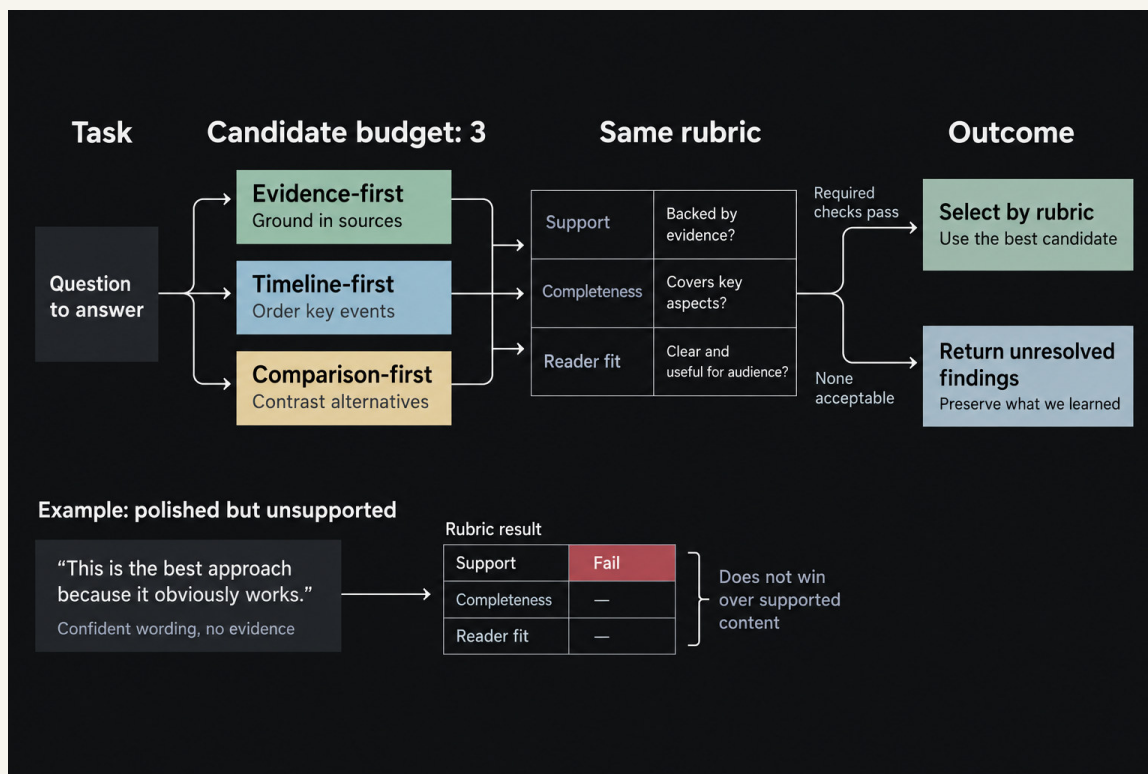


Keep a saved version while testing a candidate. Here, the revision fixes a citation but loses a required caveat, so the earlier version remains available.

Candidate selection

Sometimes it is unclear which approach is worth pursuing. The system can generate several candidates and evaluate them against the same requirements before choosing one to develop. This is often called best-of-N selection. It can explore more alternatives than repeatedly editing a single draft, at the cost of generating and checking more work.

Selection uses feedback through the choice of candidate. The next call can continue from the selected answer and its remaining findings. A candidate that fails a required check does not become acceptable merely because it scores better than the others; the program can keep searching or return the unresolved result. The criteria from Evaluation determine what the selection can meaningfully prefer.

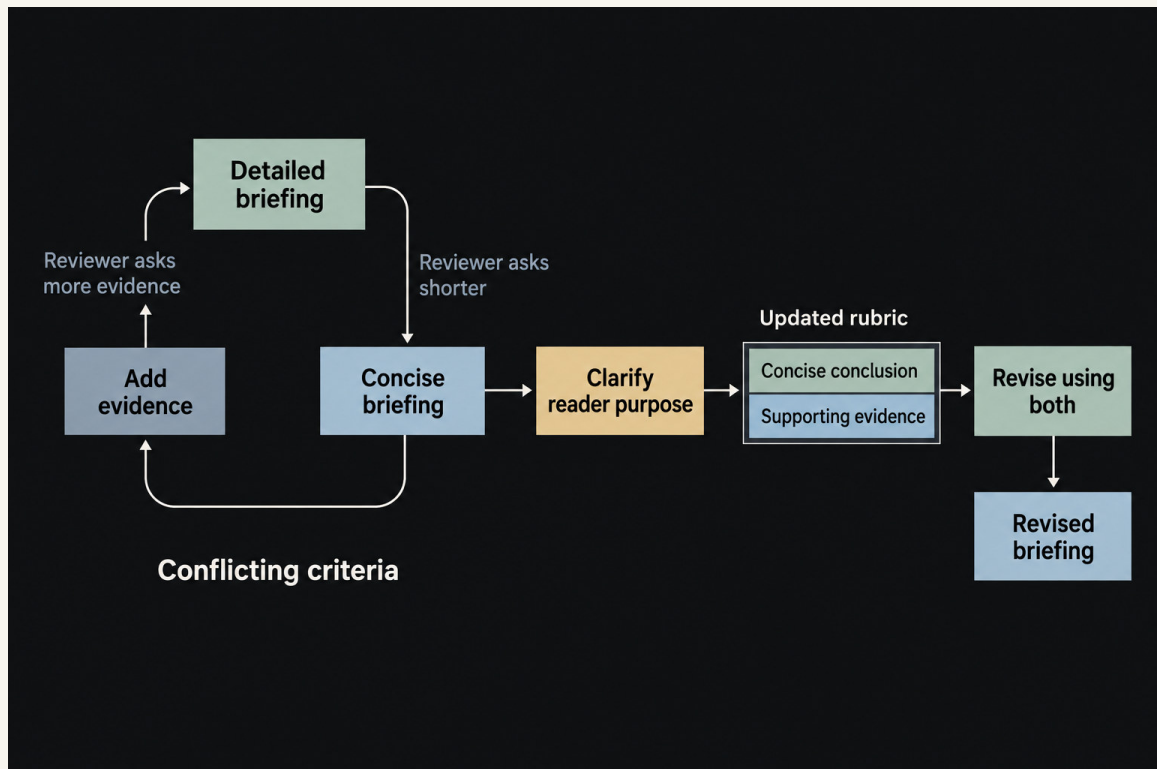


Several candidates can expand the search. The evaluator still determines what the system selects.

Convergence

The aim is for revisions to converge on work that meets the requirements. Repetition alone does not produce that outcome; a loop can also settle on a wrong answer. The model must be able to interpret the feedback and make a useful change, and the next check must be able to distinguish an improvement from a different answer.

Conflicting requirements can make the loop go back and forth. If one review asks for a shorter answer and the next asks for the evidence that was removed, each revision can undo the previous one. A concise answer with supporting detail in a separate section may satisfy both needs. The useful change is to clarify what the answer must preserve, then evaluate revisions against that stable requirement.



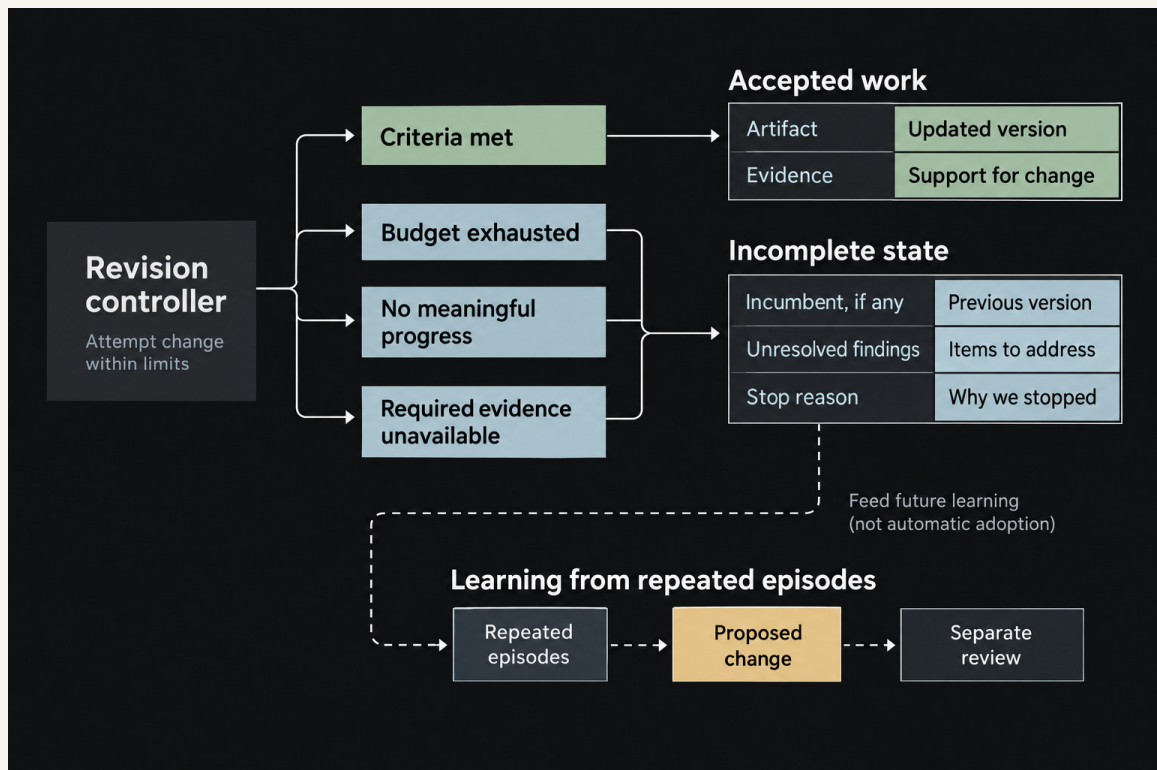
Repeated criticism does not resolve an ambiguous objective. Clarifying the intended result can change the search.

Repeated failure is information about the approach. If the same claim remains unsupported after several rewrites, another rewrite may add little. The system may need to retrieve a missing source or state that the evidence is unavailable.

Recording which findings were resolved and which persist helps it distinguish progress from repeated activity.

The loop needs a way to finish even when it cannot solve the task:

- **The checks pass:** return the result with its assessment.
- **The attempt or cost limit is reached:** return the best saved work and the unresolved findings.
- **The same problem keeps returning:** change the approach or stop the attempt.
- **Required evidence is unavailable:** record what is missing so the work can resume when it becomes available.



Acceptance, exhausted resources, stalled progress, and missing information are all legitimate ways to end an attempt.

Feedback lets the system use what happened to improve its current attempt. A later task may still repeat the same mistake. To carry the benefit forward, the system needs to preserve something reusable from the experience, such as a

better procedure for checking sources. That connection from one task to future tasks is the subject of Learning.

Learning

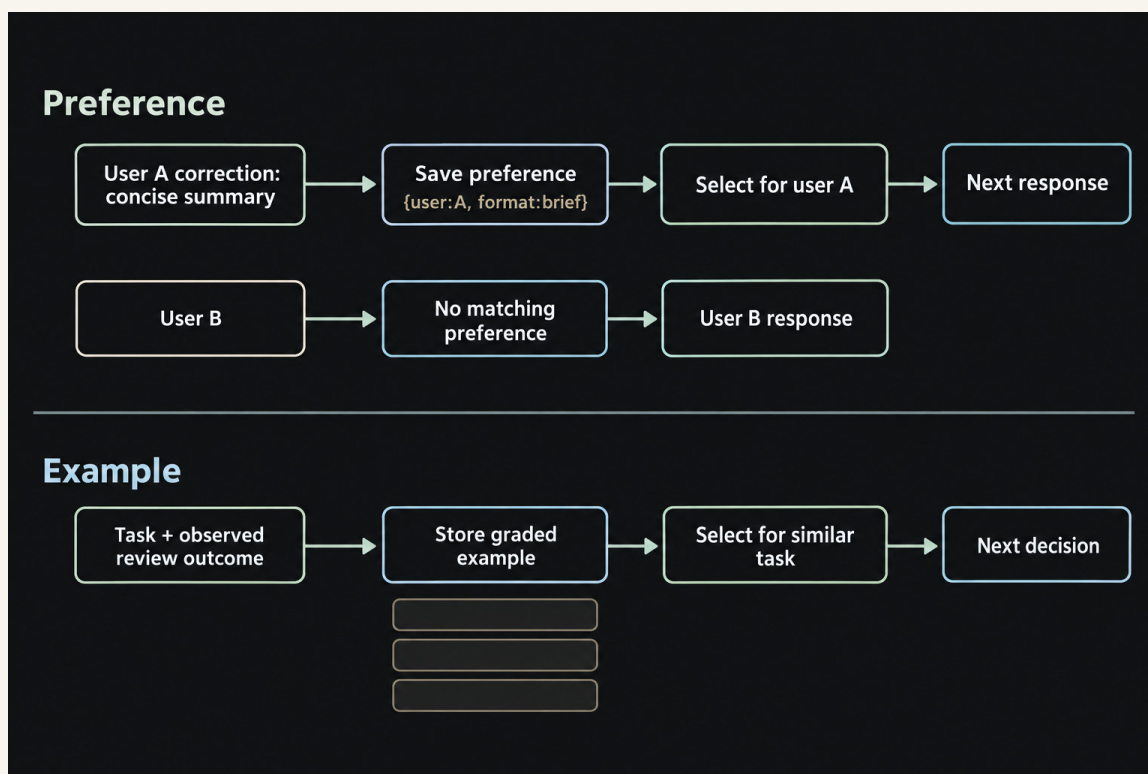
How experience improves future work.



Learning preserves a useful change so that later runs can benefit from experience.

Learning carries the benefit of experience into future tasks. The revision loop in Feedback helps an agent correct its current answer, but a new task can repeat the same mistake. The correction may exist only in a conversation that the next model call never sees. For the system to learn from that correction, something it uses on later tasks must change.

A saved procedure is one way to make that change. The next agent can load instructions that explain how to avoid the mistake, without reading the conversation in which someone discovered it. A successful answer can also become an example for similar tasks. Memory gives these lessons somewhere to persist; learning connects the experience to what the system does differently next time.



Preferences and successful examples influence later work only when they are selected and supplied at the relevant decision.

The Learning Loop

The simplest learning loop turns a useful correction into instructions for later work. In the citation example from Feedback, the agent corrects a claim after a reviewer points out that its source does not support it. The same checking procedure could help with future research answers. Preserving that procedure as a skill gives later agents a way to use the lesson.

The skill needs to explain what to do with the evidence. “Be more accurate” gives the agent little guidance. A file such as `skills/source-backed-research/SKILL.md` can describe the actual check and include the distinction that the original answer missed:

```

---
name: source-backed-research
description: Writing factual answers from retrieved sources.
---

```

For each claim drawn from a source, keep the supporting passage with its citation. Check that the passage supports what the claim says, including its dates and qualifications.

A source announcing a release next month does not support a claim that the product is available now. If support is missing, look for more evidence or state what remains uncertain.

Saving the file supplies only part of the loop. A later call must receive its contents. In a workflow dedicated to source-based research, the application can load the skill directly, as below. An agent that handles many kinds of work can instead see a catalog of skill names and descriptions, then choose which files to read. Both arrangements connect the saved procedure to a new context.

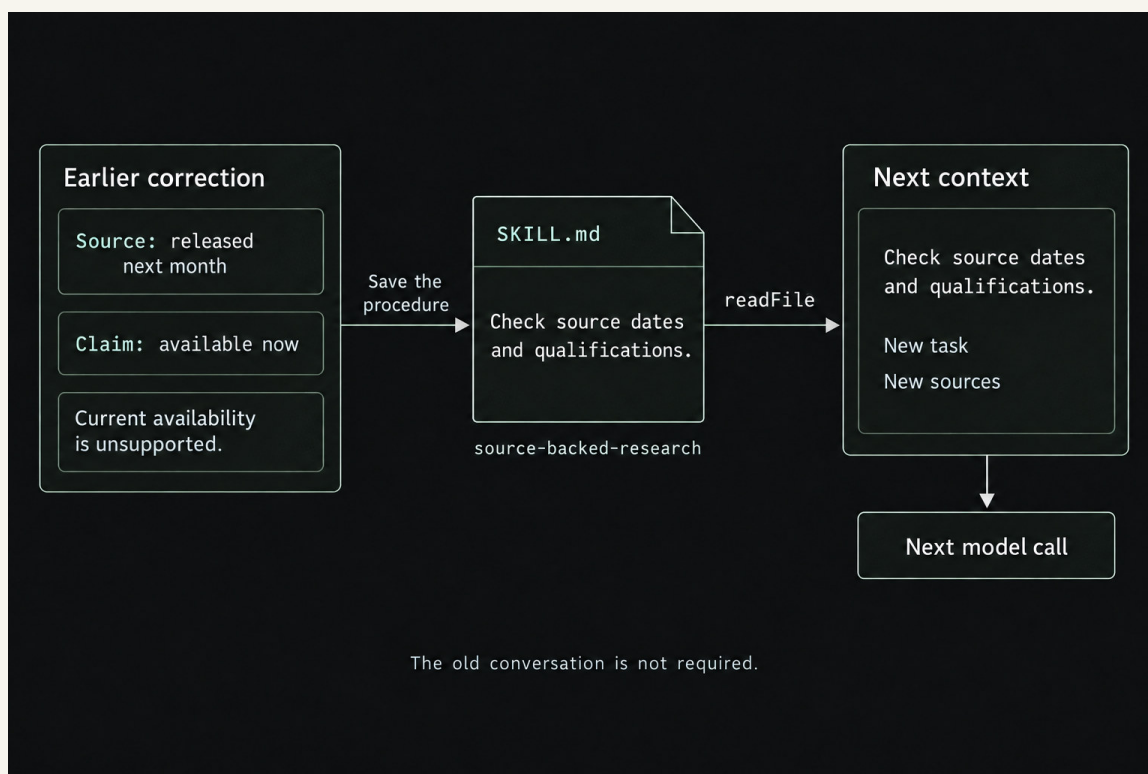
```

import { readFile } from "node:fs/promises"

// A tested procedure saved from earlier experience.
const procedure = await readFile(
  "skills/source-backed-research/SKILL.md",
  "utf8",
)

// A new task, without the old conversation.
const answer = await callModel({
  instructions: procedure,
  input: { task, sources },
})

```



The correction becomes a saved procedure. A later task reads that procedure into its context, where the source check can guide a new answer.

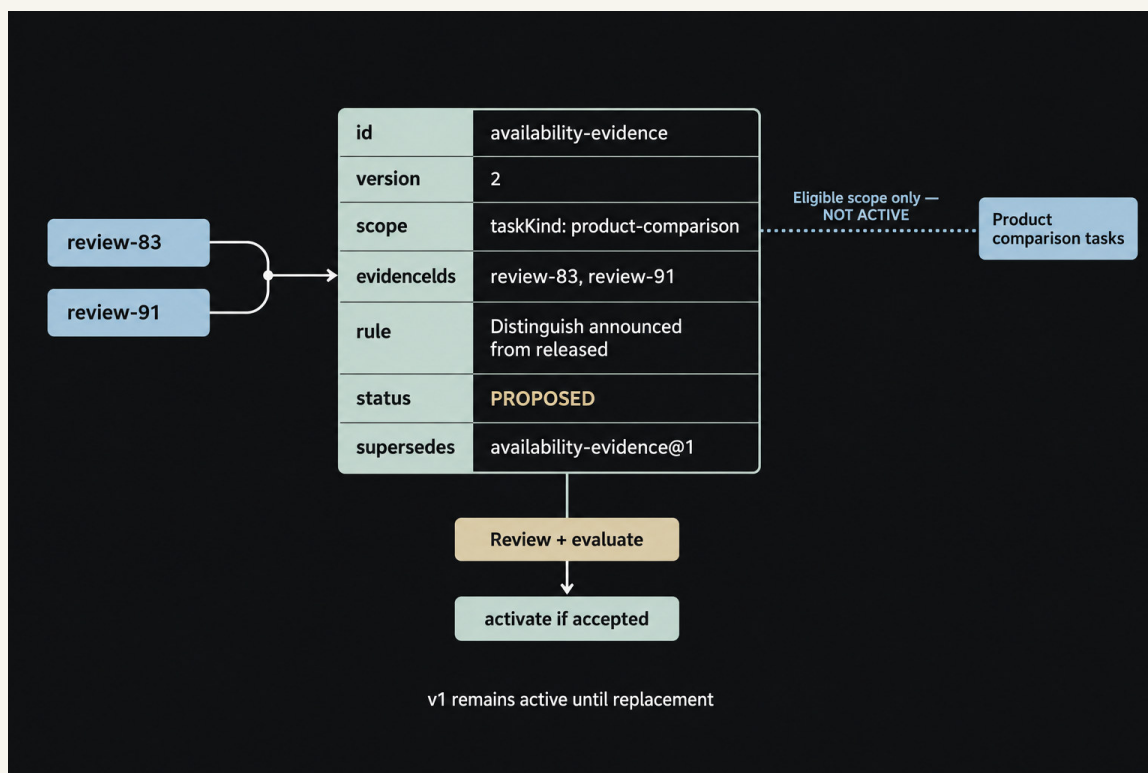
Learning from examples

An example can demonstrate how to handle a case without spelling out every instruction. A corrected answer can become a few-shot example: a sample input and desired output supplied alongside a new task. Keeping the original question and relevant source passages with the answer lets the model see how the answer follows from the evidence. A polished answer alone may teach a writing style while hiding the reasoning that made it correct.

Selecting these examples is a memory problem. A useful match shares the difficulty the agent needs help with, which may differ from the topic of the question. An example that handles conflicting sources can help with another subject where sources disagree. Retrieving several answers about the same product may add facts without demonstrating how to resolve that disagreement.

A lesson also needs a scope. An explicit preference for short weekly updates belongs with that user's weekly updates; a one-time request to shorten an answer

does not establish a preference for every future conversation. Keeping the correction and the circumstances behind it helps the system decide where it applies. The record below illustrates one way to keep that scope and evidence with a proposed rule. A skill file can carry the same information in ordinary prose.



A proposed lesson can retain its scope and supporting observations. The application must use that scope when deciding where the lesson applies.

Changes to the harness

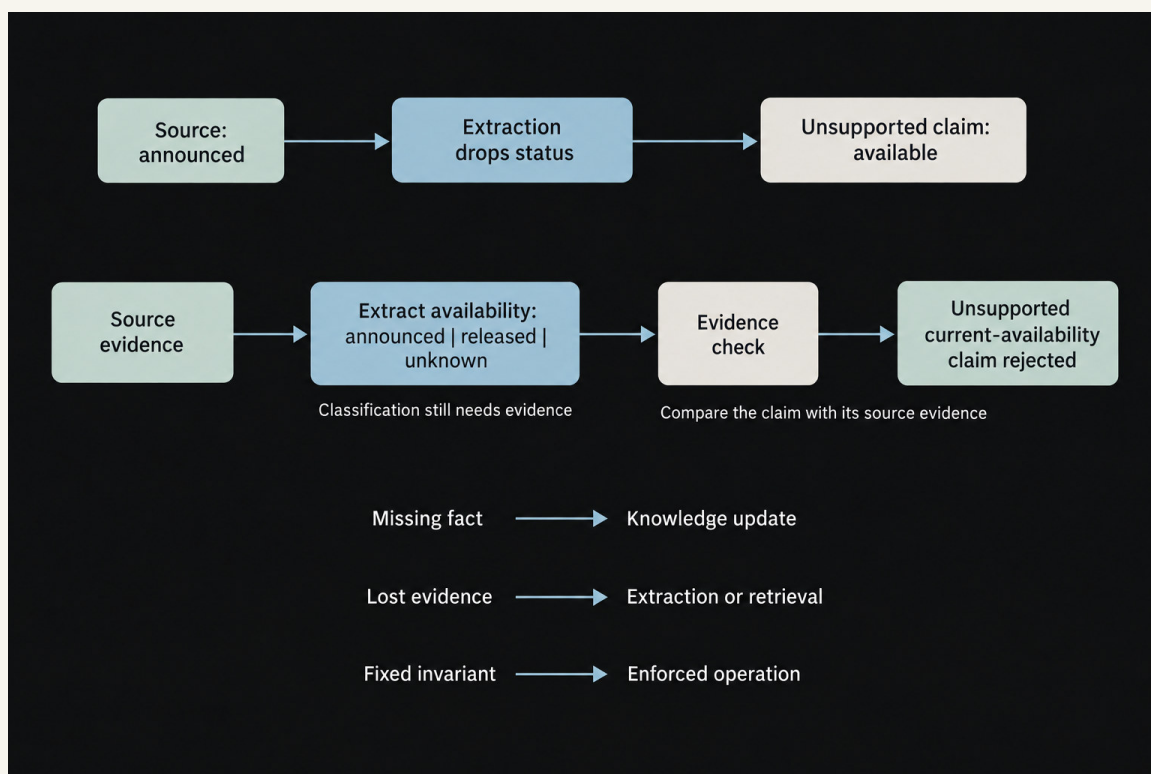
The harness is the software around the model. It determines what information reaches each call and what happens to the result, so a recurring error may require a change there. In the citation example, instructions to check release status will do little if an earlier extraction step has already removed the dates and qualifications from every source.

An improved extraction step could preserve availability: "announced" | "released" | "unknown" alongside the supporting passage. Later calls would then receive the distinction they previously lacked. The field does not prove that the

classification is right; the passage makes it possible to check. The lasting improvement comes from preserving evidence before another answer needs it.

Other parts of the harness can change in response to experience:

- **Retrieval:** a search that repeatedly misses relevant documents can lead to better indexing or a different ranking method.
- **Routing:** tasks that need calculations can go through a code-execution step after direct answers prove unreliable.
- **Tool implementations:** an overwritten document can reveal the need for a save that writes only if the stored version still matches the version the agent read. The check and write happen as one operation.

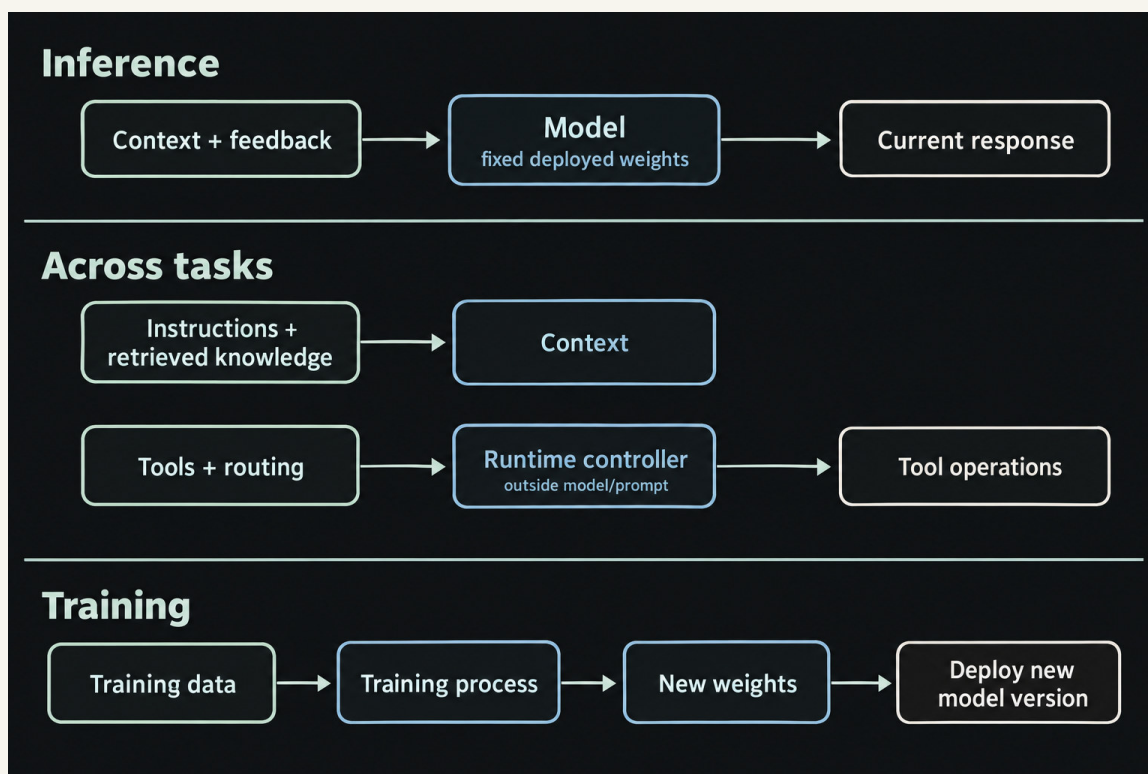


Preserving a missing distinction in extraction or retrieval can prevent the same loss of evidence in later runs.

Model training

All the changes so far can work while the model's weights stay fixed. Ordinary inference uses the parameters the model already has; correcting an answer in a conversation does not itself update those parameters. A later call benefits because the application supplies different information or runs different code.

Training provides another place for experience to go. Fine-tuning updates model parameters using training examples. A collection of successful task inputs and outputs can therefore serve different purposes: the application can retrieve selected examples into context, or a training process can use them to change the model. Context examples are easy to inspect and replace. Training can reduce the need to restate a behavior on every call, but takes a separate update process and does not guarantee that the model learns the intended lesson.



Context adaptation, persistent system updates, and parameter training change different parts of an agentic system.

Automated reflection

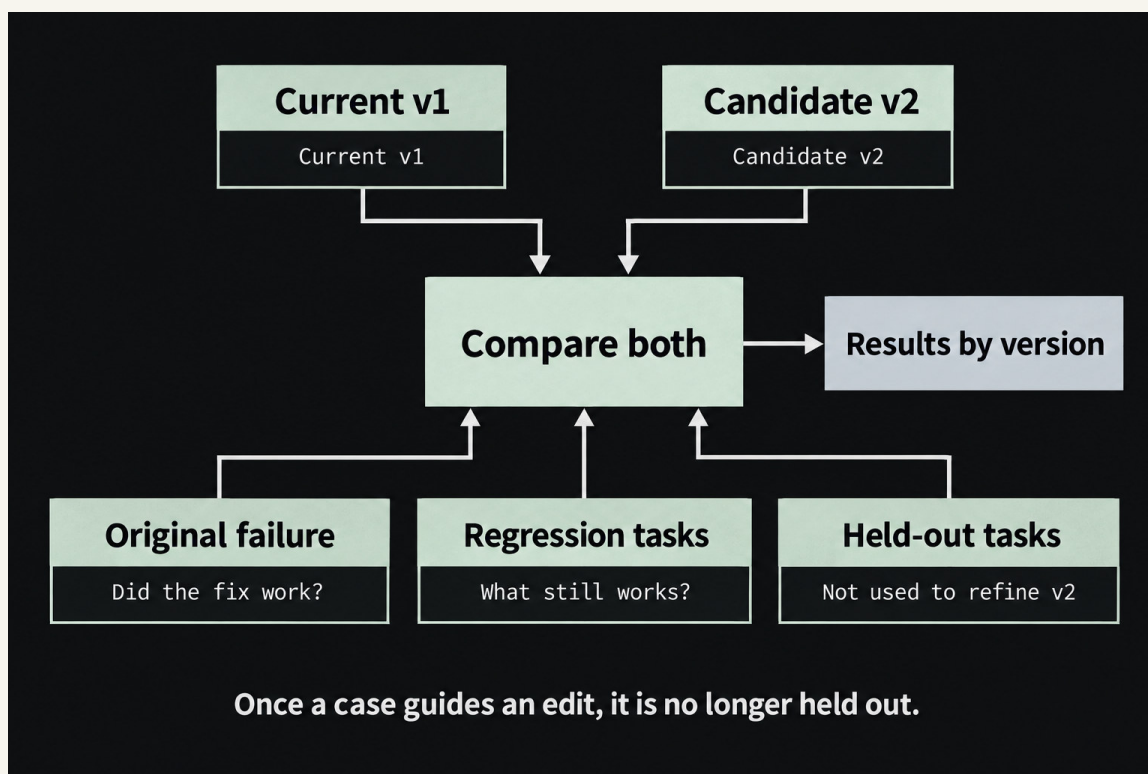
A developer who reads an agent's failures and improves its skill file is already completing a learning loop for the system. The same work can become an agent task. A reflection job can examine saved attempts and their evaluations, identify a recurring problem, and propose a change to the relevant skill or tool. A heartbeat can schedule this review periodically, using the trigger and context-reconstruction mechanisms from Autonomy.

The reflection call needs evidence of what happened. The final answer and a low score may not reveal whether the agent lacked a source or ignored a source it had. The task, tool results, and evaluator's findings help distinguish those causes. Useful reflection produces a specific proposed change with the observations that support it. A fluent explanation of a failure is still a hypothesis about its cause.

Testing a lesson

A change that fixes one answer can make other answers worse. A rule inferred from a missing citation might demand citations even for a task that only asks the agent to rewrite supplied text. Evaluation checks whether the proposed lesson helps where it should and leaves other useful behavior intact:

- **The original failure** checks whether the change addresses the problem that prompted it.
- **Previously successful tasks** check for regressions, including within the same kind of work.
- **Held-out tasks** test the change on cases that were not used to write or refine it. Once a case guides an edit, it no longer serves as an independent check.



Compare the old and new procedures on the original failure, previously successful tasks, and cases that did not shape the change. Once a case guides an edit, it is no longer held out.

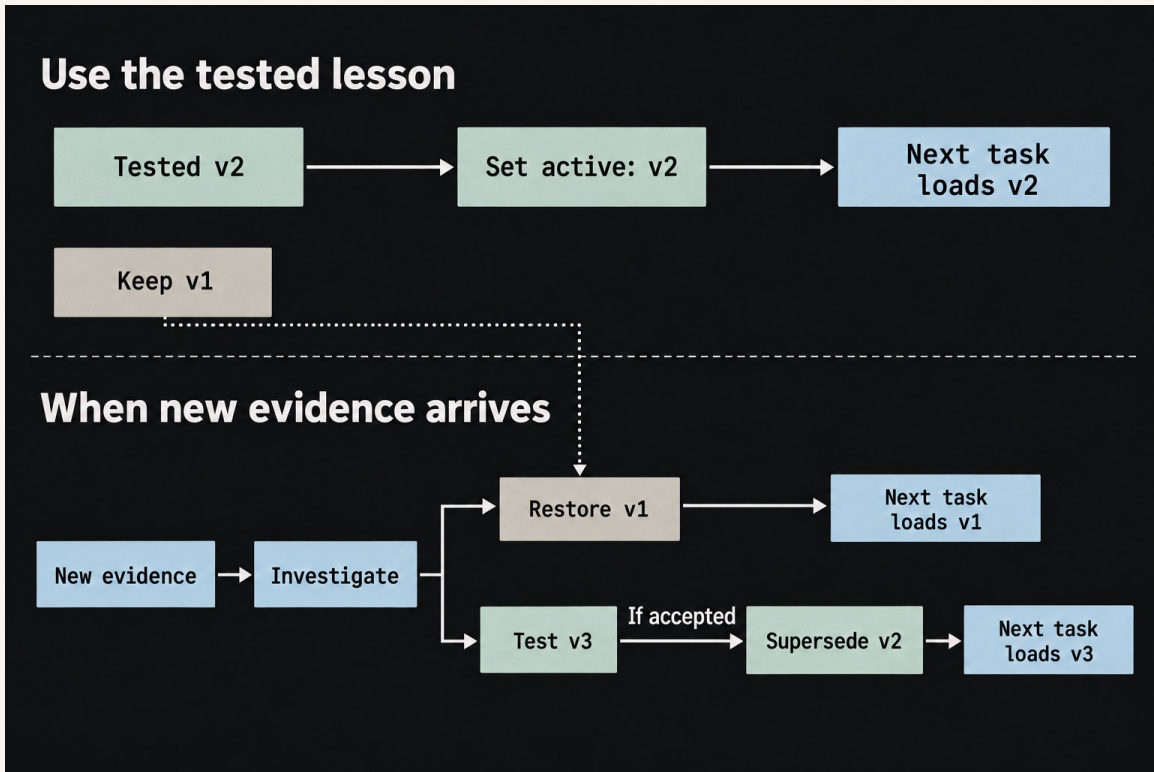
Comparing the old and new procedure on the same tasks makes the effect easier to judge. Keeping the model and other settings fixed helps separate the proposed change from other causes of improvement. Because model outputs vary, a single better answer may not settle the comparison. The reliability measurements from Evaluation apply here too.

Accumulated lessons

Using a tested lesson requires an actual update: the application must load the revised skill or run the changed code. Keeping the previous version makes it possible to undo a change if later results expose a problem the tests missed. A version identifier also lets us connect a result to the procedure that produced it.

As lessons accumulate, the context limit returns. Loading every instruction into every task fills the window with irrelevant advice, and old rules can conflict with new ones. A skill catalog lets the model choose a relevant procedure without reading the whole collection. Updating an existing skill keeps related guidance

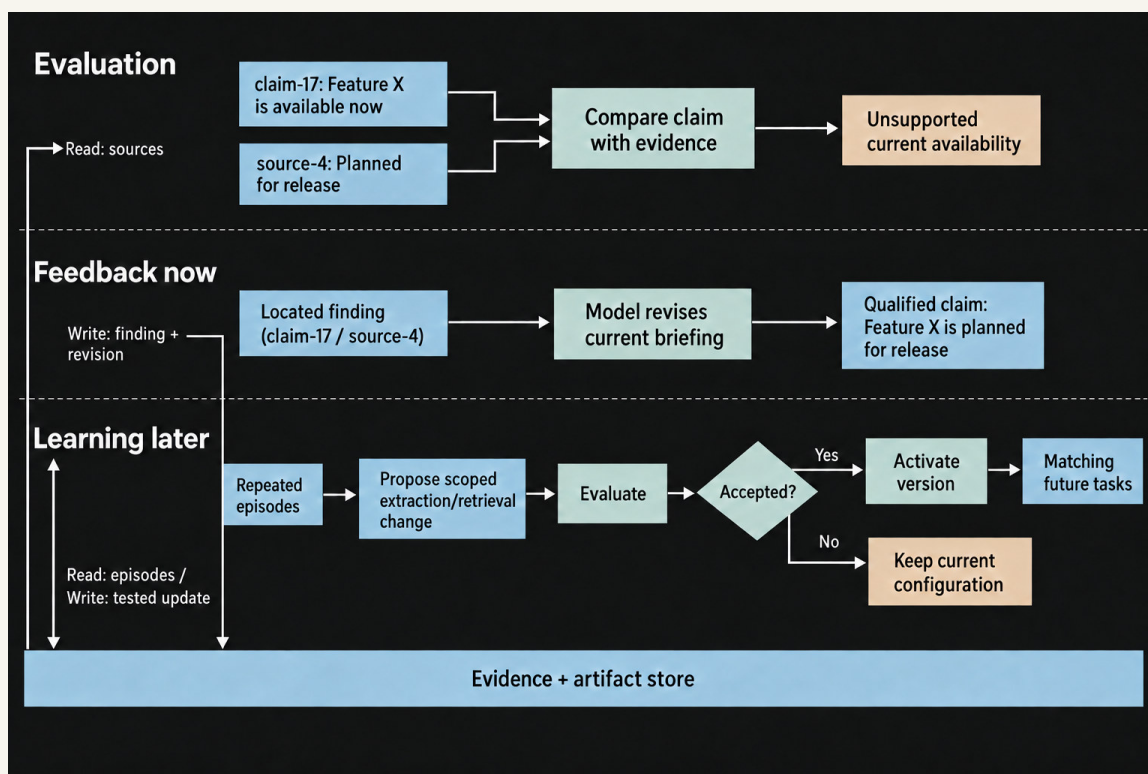
together; replacing an obsolete instruction prevents two versions from competing in the same context. The system can keep the old evidence in its history without continuing to use the old rule.



Later tasks must load the tested procedure. Keeping earlier versions allows the system to replace a lesson or restore its predecessor when evidence warrants a change.

Learning systems

Evaluation, feedback, and learning connect at the same piece of work. Evaluation identifies an unsupported claim. Feedback helps the agent correct that answer. Learning uses the discovery to improve the procedure or information available to later tasks. Each connection has to exist in the program; storing a trace or writing a reflection does not create the next connection automatically.



Evaluation identifies the problem, feedback repairs this attempt, and learning tests a change for later attempts.

The book began with a model call that receives task information through a finite context window. That constraint keeps returning as we build larger systems. Dividing work among agents gives each one a focused context, but creates a need to share results through messages and artifacts. Continuing work across separate runs requires stored state and a way to reconstruct it. Learning makes those stores and procedures subject to improvement through experience.

These elements give us a way to reason about designs that have yet to become familiar patterns. A new arrangement of agents still has to get the right information to each decision and turn the decisions into useful work. Once those connections are visible, we can explain why a design works, locate what it is missing, and build the structures the task actually needs.

Further reading

The distinctions in this book can be explored through research and working implementations. These sources give more detailed accounts of particular mechanisms and their limitations.

- [Lost in the Middle](#) studies how the position of relevant information affected the models and retrieval tasks tested. It supports the context chapter's treatment of placement as an empirical question.
- [Anthropic's Building effective agents](#) explains workflows, agent loops, and composable patterns. Its architectural distinctions are useful beyond the tooling available when it was written.
- [Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena](#) examines model-based evaluation, including position, verbosity, and self-enhancement biases.
- [Anthropic's Demystifying evals for AI agents](#) develops the distinction between an execution transcript and its outcome, alongside repeated trials and human calibration.
- [The Model Context Protocol architecture](#) explains how applications connect to servers providing tools and other context. The Agent Skills format describes reusable instructions and resources loaded when needed.
- [Temporal's documentation on durable execution and Activities](#) gives a concrete implementation of persisted execution history and retryable work.

Implementation references

[Vercel AI SDK Core](#) is a useful starting point for TypeScript examples of tools and bounded model loops. [LangGraph](#) makes state and execution graphs explicit. These libraries package several of the elements together; their interfaces can be understood in terms of the context, control flow, and state they manage.

About this edition

Written by William Chen at Idyllic Labs. This second edition develops the original Elements of Agentic System Design into a continuous guide. It is a working draft. AI tools assist with research and drafting.

The text is available under Creative Commons Attribution 4.0. Comments and corrections can be sent to william@idylliclabs.com.