ENGINEERING PRINCIPLES

Building Effective AI Agents

A vendor-neutral field guide to the patterns, the tradeoffs, and the reasoning behind each choice, written for the leader who will brief the team, make the architecture calls, and own the roadmap of an agent product.

Maximize the prompt, minimize the code.

Author	X-Arc
Date	June 2026
Audience	Product and engineering leaders

ABSTRACT

An agent that performs well in a demonstration will often behave differently once it is deployed and left to run on its own, and the reason is rarely that the model became less capable. The conditions changed, because the work is now unattended, repeated, and chained across many steps, and the gap that opens between a capable model and a dependable system is where most agent projects fail. This guide is about closing that gap from first principles. It rests on a single engineering bet, that you should maximize the prompt and minimize the code, putting behavior in language the model reads and reserving code for the small set of guarantees the model must not be trusted with. The bet is argued rather than asserted, and its second half is what keeps it honest, because the discipline of minimal code draws a hard, fail-closed boundary around the outcomes that can tolerate no variance, the ones that are irreversible, financial, cross-tenant, or able to fabricate trust. Every pattern that follows, from the harness that separates who owns the loop from who owns the decisions, through tools, retrieval, memory, human approval, streaming, observability, evaluation, release safety, and multi-agent work, is one application of that same boundary. The guide is vendor-neutral and none of the specifics are proprietary.

How to read this

The guide is written for the person who will brief engineers, make architecture calls, and own the roadmap, rather than for someone who needs to copy code, and every section keeps the same shape so the reasoning stays in view. Each one opens with the mental model, the idea you should be able to draw on a whiteboard, then moves to what it means in practice, the concrete pattern with light pseudocode where it earns its place, and closes on the tradeoff, what you pay and the conditions under which paying it is worth it. One idea sits underneath all of them, and if you take away nothing else it is this, that you maximize the prompt and minimize the code, putting behavior in language and letting the model decide while reserving code for the small set of guarantees the model must not be trusted with. The first section is the argument for why that holds, so that it reads as a conclusion rather than a slogan.

1 The core bet: max prompt, min code

The mental model

Traditional software encodes behavior in control flow, in branches and state machines and orchestration graphs, and with a capable model that instinct becomes a trap, because hard-coded logic is scaffolding the model will outgrow. The discipline is to write the least code that still keeps the system safe and to express everything else, what to do, in what order, how to recover, and how to phrase the result, in prose the model reads. Prose is the steering wheel and code is the chassis.

Why it holds

Maximizing the prompt is an engineering conclusion rather than a matter of taste, and it follows from three measurable properties of capable models. The first is that general methods which scale with computation beat handcrafted ones, by a large margin, which is the practitioner reading of Sutton's Bitter Lesson[1], and applied to agents it means a prompt is a general control surface that improves on its own as the model improves, whereas hardcoded logic is handcrafted knowledge the next model outgrows, so you spend behavior on the surface that compounds rather than the one that bottlenecks[2]. The second is that in a non-deterministic system code is the high-variance control surface, which is the counter-intuitive part and is observed rather than assumed, since minor changes to code cascade into large behavioral changes that make complex agents remarkably difficult to write in code[3], while prose is the comparatively low-variance and inspectable surface, where a behavior change is a paragraph you can read and diff. The third is that attention is a finite budget, so context engineering is the work of curating the smallest set of high-signal tokens that maximize the likelihood of the outcome you want[4], which means you are allocating a scarce resource with diminishing returns rather than simply writing instructions.

Maximizing the prompt is not granting free will

The common misreading is that maximizing the prompt means letting the model do whatever it wants, and the second half of the bet, minimizing the code, is precisely the mechanism that prevents it. The discipline is to partition every decision the system makes by a single question, which is how much variance the outcome can tolerate. Outcomes that can tolerate none, the ones that are irreversible, financial, cross-tenant, or able to fabricate trust, are guaranteed in code, fail-closed, with the model not trusted, while outcomes that are advisory, interpretive, or recoverable are delegated to the model and steered by prose. This is bounded autonomy rather than free will, because the model is free only over the set where a wrong answer is survivable, and code enforces hard invariants over the set where it is not. Minimal code does not mean no code, it means code appears in exactly one place, where a model mistake would cross that line.

Table 1. The boundary is the framework. Every section that follows is one application of this single partition between what the model decides and what code guarantees.

The model decides `prose-steered, trusted`	**Code guarantees** `fail-closed, model not trusted`
Which tools to call, and in what order	A scoped identifier is real-shaped before it touches data
How to interpret the data and what to recommend	A citation renders only when it maps to a real retrieved result
What to assume when the user is silent	Identity is bound to the authenticated session, denied on failure
How to recover from an error and how to phrase the answer	Every state-changing action passes a human gate and a schema check
When to delegate and when to refuse	The write lands on the server-resolved account, not the one the model named

What it looks like in a healthy codebase

The clearest signal of a healthy agent codebase is that the prompt files dwarf the code that ships them. In one production system the agent's instruction document ran to roughly twelve hundred lines of structured prose while the code that assembled and sent it was a handful of small functions, and the behaviors a conventional system would hardcode all lived in language instead, including which tool to call and in what order, when to fetch a knowledge source and how to reconcile conflicting ones, what counts as healthy and what is a red flag, what to assume on silence, how to recover from a timeout, and the output format, citation style, and refusal stance. A telling habit follows from this, which is that behavior bugs get fixed by editing prose rather than code, so when a user reports that the agent defaulted to the wrong scope the fix is three sentences added to the tool's description and the instruction file, together with a test asserting that the description still contains that sentence, and no new branch is introduced. The deliverable of a bug fix is often a paragraph.

> **Principle.** Behavior lives in prose, and code is a thin, safe substrate. Delegate everything advisory, interpretive, or recoverable to the model, and enforce in code only what is irreversible, financial, cross-tenant, or able to fabricate trust.

2 Workflow versus agent, and the augmented LLM

The mental model

Not everything should be an agent, and the line the industry now uses is a clean one[5], where a workflow is a set of language models and tools orchestrated through predefined code paths, so that you decide the steps, and an agent is a system where the model dynamically directs its own process and tool use, so that the model decides the steps. The atom underneath both is the augmented model, a model paired with retrieval, tools, and memory, and the right discipline is to build that well first and then escalate only as far up the autonomy ladder as the problem genuinely demands.

Figure 1. The autonomy ladder. The left end is cheapest and most predictable, the right end is most capable and least predictable, and each rung upward should be earned by the problem rather than assumed.

What it means in practice

The most quoted line from the source is a product directive rather than a technical one, that you should find the simplest solution that works and increase complexity only when it is needed, because the most successful implementations were not the most complex but the most well-suited[5]. Use a workflow when the path is knowable and you want predictability, low latency, and low cost, and reach for an agent when it is difficult or impossible to predict the number of steps and you cannot hardcode a fixed path, since an agent trades latency and cost for the ability to handle open-ended work, which is a trade to make deliberately and per feature rather than as a default. A single product can sit at both altitudes at once, with a deterministic intake and routing layer handing off to an agentic core only for the part that is genuinely open-ended, so that you never have to bet the whole product on one setting.

> **Principle.** Start with the simplest thing that works and add autonomy only where the problem is genuinely open-ended, because the agent setting is the most expensive position on the dial and should not be left there by default.

3 The harness: separate the loop owner from the decision owner

This is the single most important architectural decision in the guide, and getting it right is what lets the rest compose cleanly.

The mental model

A turn is a loop, in which the harness builds the prompt, asks the model, executes the tools the model requested, feeds the results back, and repeats until the model reports that it is done, and the loop's exit condition is literally the model saying that it is done[6], which is the same loop Anthropic frames as gathering context, taking action, verifying the work, and repeating[7]. The critical move is to separate who owns the loop from who owns the decisions, so that the harness owns the turn, meaning it assembles context, dispatches, streams output, persists, and enforces limits, while the model owns the loop, meaning it decides which tool to call, when, how many times, and when to stop. In a well-built harness the only loop your own code runs is the one that iterates the model's output events, because you iterate what the model produces and you do not drive what it decides.

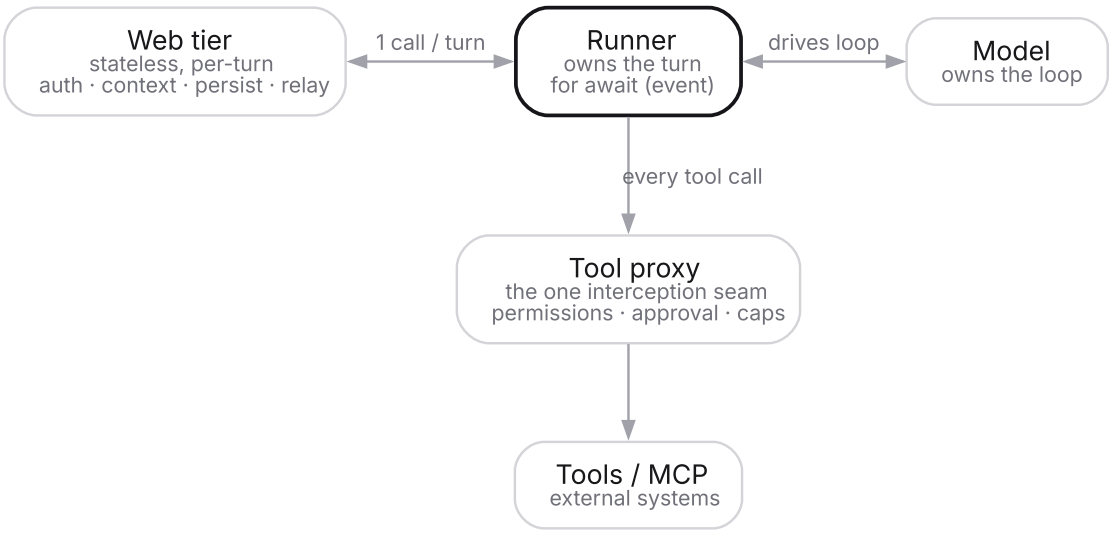

Figure 2. Three seams. A stateless web tier, the runner that owns the turn, and a single tool proxy that is the one place the model touches the world.

Four patterns that carry their weight

The first pattern is that the web tier is a context assembler and a relay and never a loop owner, so it does authentication, gating, and context assembly, then dispatches one request to the runner and streams the result back, which keeps the user-facing tier stateless and horizontally scalable while the long-lived loop lives in a service built for it. The second is that there is exactly one interception seam between the model and the world, so every tool call routes through a single proxy, and that proxy rather than a scatter of checks is where you enforce permissions, approvals, and response-size limits, with the useful consequence that you can disable the model SDK's own approval prompts and gate everything at this one seam, since from the model's point of view a denied tool simply returns a structured error and it cannot tell that it was gated. The third is that you persist a thread handle rather than a transcript, because modern agent SDKs hold conversation state server-side, so to resume you store only the thread identifier keyed to the conversation and skip re-sending history, and to branch by edit-

ing a past message you clear the handle and force a clean replay, which is dramatically cheaper than re-streaming the full history on every turn. The fourth is that one turn-driver serves every origin, so interactive chat, a scheduled background run, and a channel integration all share the same route discriminated by a single origin field, which flips identity and persistence and, importantly, collapses the human approval step where no human is present, because a scheduled run maps automatic approval onto the same permission machinery that interactive chat uses for human clicks. You build the turn engine once and vary it by origin.

The tradeoff

This separation costs you a network hop and a serialization boundary between the web tier and the runner, and you pay it gladly, because it is what lets the stateless front end scale, the long-running loop survive, and the security gate remain a single auditable seam. The failure mode of not doing it is a monolithic route that owns authentication, the model loop, tool execution, and persistence at once, which is impossible to scale, secure, or reason about.

> **Principle.** The harness owns the turn and the model owns the loop. Put exactly one interception seam between the model and the world, resume by thread handle rather than by replay, and build the turn engine once so you can vary it by origin.

4 Tools: the model's hands

The mental model

A tool is the only way an agent affects or perceives anything beyond text, and the most precise framing is that a tool is a contract between deterministic systems and non-deterministic agents[8]. It has three parts, and all three of them are load-bearing prompt surfaces rather than plumbing.

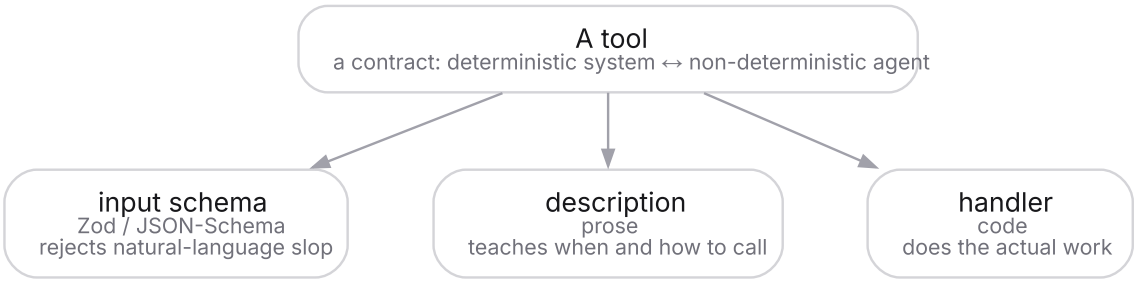

Figure 3. A tool is three load-bearing surfaces. The schema redirects the model, the description teaches it, and the handler does the work.

The mistake beginners make is to treat the description as a label and the schema as mere validation, when in a good agent the description teaches, covering when to use the tool, when not to, the caveats, and the exact wire shape, and the schema redirects, rejecting natural-lan-

guage slop and steering the model away from confusing an identifier with a display name.

What it means in practice

The description is part of the prompt, so you should spend tokens on it, and a good tool description often runs to twenty lines or more, with a note on the data horizon, the exact input shape, an explicit block of caveats, and a block describing when the tool should be used, because production traces show models burning far more tokens on retry-after-empty loops caused by a vague description than a verbose description ever costs, and the schema's own field descriptions teach as well, for instance by telling the model to sort by a bare metric key rather than a display column. Errors should be returned as data the model can recover from, so you never throw a raw stack trace or an upstream error at the model and instead return a structured envelope.

```
{ "success": false,
  "error":   "UPSTREAM_TIMEOUT",
  "message": "The report took too long. Try again, or narrow the query (fewer items).",
  "hint":    "The write MAY have applied. LIST to verify before retrying.",
  "retryable": true }
```

Error messages are teaching moments rather than notifications[8], and three refinements matter in production, where you split codes by fault so that a client timeout, an upstream timeout, and an internal error are distinct and the model and your dashboards attribute them correctly, you make the hint safety-aware for non-idempotent writes by telling the model to verify before retrying rather than retrying blindly, and you keep the raw error server-side for forensics while handing the model the clean version. The Model Context Protocol is the integration standard worth adopting[9], because it solves the problem of every model needing a bespoke connector for every tool, and two production notes apply, that you build the tool server stateless and per-request behind a load balancer so that authentication and tenant context cannot cross-contaminate between users, and that you treat tool annotations such as the read-only and destructive hints as functional rather than cosmetic, since they drive the host's approval interface and on some clients decide whether a call executes at all. Steering should travel with the tool, because some hosts that connect over the protocol ignore your system prompt entirely and read only the tool list, so the domain rules that must always hold, the defaults, the sequencing prerequisites, and the instruction to confirm before writing, get compiled into the tool descriptions themselves and then mirrored in the system prompt and any skill files so that they stay in lock-step. Finally you gate tools at three layers rather than one, at registration, where you decide which tools even exist this turn by authentication scope, tenant state, and feature flag, at the per-call permission resolved at the proxy, and at the feature flags read at call time, and a

convention that pays off is to encode the fact that a tool writes into its name, with a `propose_` prefix, so that the default permission rule becomes a single line, that any unknown or `propose_` tool needs approval, which fails closed for any tool someone adds later.

> **Principle.** The description and the schema are prompt surfaces, so invest in them, return errors the model can reason about, standardize on the Model Context Protocol, build the server stateless, and make steering travel with the tool so that it survives hosts that ignore your prompt.

5 Knowledge and retrieval: a tool, not a prefix

The mental model

The naive retrieval pipeline embeds everything, retrieves the top matches, and stuffs the chunks into the prompt, which makes the whole system hostage to a single retrieval call, so that when retrieval is mediocre even a strong model cannot recover, because it never sees the right context and has no way to ask for more. The shift the field has made, and the one we made in production, is to stop pre-injecting a fixed set of chunks and instead give the model a search tool it calls on demand and iteratively, deciding for itself when it has enough, which is the same logic as the harness, letting the model drive[10].

The recurring decision, inject or fetch

The decision recurs across knowledge, skills, and memory, and it is the same decision each time, so getting it right is most of the work.

Table 2. The inject-or-fetch decision. The unifying rule is to inject only what is small, specific to the turn, and always relevant, and to give the model a tool for everything that is large or situational.

Concern	Where it lives	Why
Large knowledge corpus	`tool`	The corpus is large and the relevant slice varies per question, so you pay tokens only when they are needed.
Curated must-cite knowledge	`tool, mandated`	The prompt forces the call as policy while the tool supplies the content.
Large reusable skill	`file, on demand`	The client pulls it only when the topic calls for it, through progressive disclosure.
Small per-user playbook	`prompt, per turn`	It is cheap, specific, and needed every time it applies.
Long-term memory	`tool recall`	A content-free steering note avoids a per-turn token tax and keeps recall inspectable.

What it means in practice

You over-fetch and then re-rank in code, because the vector score alone is not the final ranking, so a reliable pattern is to fetch around fifty candidates and then run a deterministic re-rank that deduplicates near-identical content, weights by source authority, and penalizes stale versions, keeping the best handful, so that the model receives a small high-quality set rather than fifty noisy chunks. You do not retrieve what already fits in the context window, since a useful rule of thumb is that a knowledge base which fits in roughly two hundred thousand tokens, about five hundred pages, can simply be placed in the prompt, which makes retrieval a scaling tool rather than a default[11], and when you do chunk you guard against naive chunking destroying meaning, because a chunk reading that revenue grew three percent loses which company and which quarter, so you prepend a short chunk-specific context note before indexing and cut retrieval failures by a third or more. You mandate usage while supplying content through the tool, so that when grounded answers are the product the system prompt can carry a hard rule that every claim must be grounded in a knowledge-tool result from the same turn and that fabricated citations are out of scope, while the knowledge itself stays tool-fetched and never injected, and one sharp lesson is that a blanket rule to ground everything causes over-pull, where the model calls the knowledge tool even on operational turns that make no factual claim, so the fix is a carve-out rather than a relaxation, since grounding gates advice and not actions. Finally you make citations opaque identifiers that code verifies, so the model cites with an opaque protocol link whose identifier came from a tool result rather than a raw URL, and the client resolves those identifiers against what was actually retrieved this turn, rendering a real card when the identifier is real and degrading to plain text when it is not, which is a small instance of the boundary, since the model decides what to cite while code guarantees that the citation is real.

> **Principle.** Retrieval is a tool the model calls rather than a prefix you inject. Over-fetch and re-rank, do not retrieve what already fits in context, mandate grounding for claims rather than actions, and make citations opaque identifiers that code verifies before they reach the user.

6 Memory: recall over injection

The mental model

The instinct is to build a memory table and prepend its contents to every system prompt, and you should resist it, because the better pattern is the same as retrieval, giving the model memory tools to save, search, and read, and letting it recall on demand, while injecting only a tiny content-free note that tells the model the tools exist and when to use them. Recall is therefore tool-driven, so memory is not stuffed into every system prompt, and it becomes explicit, inspectable, and deletable rather than passive background magic.

What it means in practice

You inject the announcement of the capability rather than the data, in a few sentences that tell the model it has durable memory, that it should save salient facts proactively, and that it should search the whole store before concluding it has nothing, since that last clause matters, because without it models scope their search too narrowly and wrongly conclude that there is no memory. Writes can be proactive and ungated while recall stays at the model's discretion, because saving a fact is cheap and safe whereas pulling a fact is the model's call when it is relevant. You isolate memory by an identity you derive on the server, never one the client supplies, for example a hash of the authenticated user's email under an issuer namespace, because a store keyed by a client-set value is a cross-tenant leak waiting to happen, which is one of the guarantees that belongs on the code side of the boundary. You also couple the runtime gate and the prompt gate, with two flags that flip together, one that exposes the memory tools and one that injects the steering note, so that you never tell the model to use a tool that is not wired up and never wire up a tool the model is never told about.

> **Principle.** Memory is tool-recall rather than per-turn injection. Announce the capability, store the data outside the prompt, isolate it by a server-derived identity, and keep it inspectable and deletable.

7 Human in the loop: pause the tool call, not the agent

The mental model

Any consequential action, a write, a spend, or an irreversible change, needs a human gate, and the elegant way to build it is not to pause the agent loop but to pause the tool call. At the interception seam, when a call needs approval, the proxy waits on a promise that resolves only when a human decides, so that to the model the tool simply took a while and then returned either a result or a structured denied envelope, and the loop never knows that it stopped. The canonical decision set, drawn from both the major agent SDKs[12][13], is to approve, edit, reject, or respond, and the invariant that matters is that the interrupt happens before the side effect rather than after it.

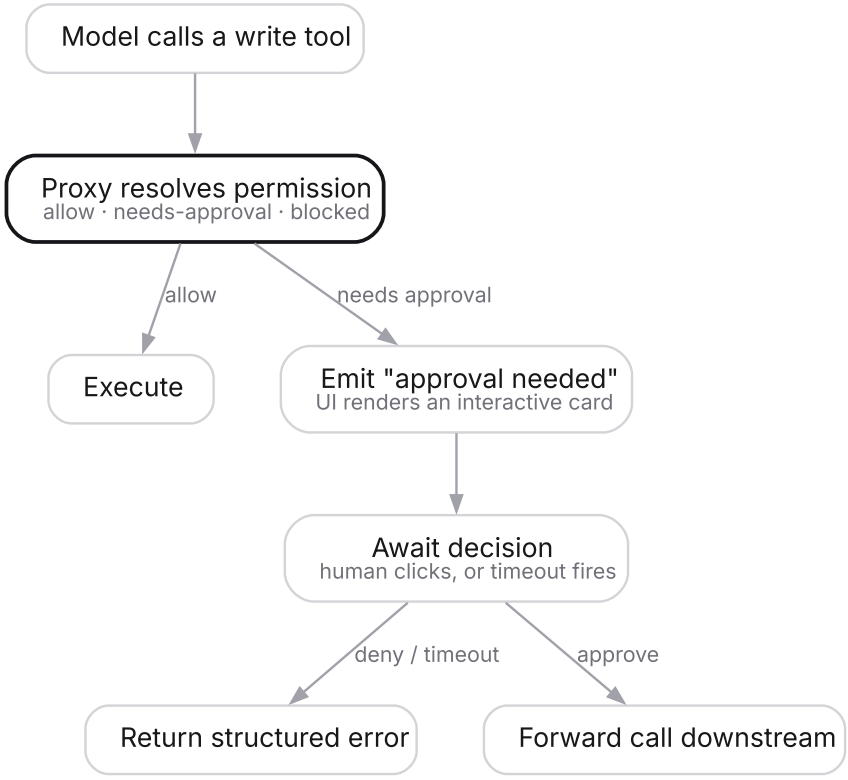

Figure 4. Human in the loop. The tool call pauses on a promise at the proxy, and the agent loop never knows that it stopped.

Production refinements worth copying

You make approval a first-class interface element rather than a question in prose, because the agent should act rather than ask whether it should, so the approval card is the consent surface, which keeps the agent's output clean and the action unambiguous and clickable. A timeout becomes an automatic denial returned as a structured error, since a pending approval must never hang the turn forever, and a real bug here was a stale timeout constant set well below the actual tool timeout, which silently auto-denied valid actions, so the two numbers must be kept in sync and the dependency commented. You keep two scopes of remembering a decision, where a per-tool and per-conversation always-allow persists on the server so that it survives a reload, while a session-wide automatic approval is client state gated behind a one-time consent. The safest action is the default keypress, so the plain return key allows once, the modified return key allows always, and the escape key denies. Finally you serialize the paused state for asynchronous decisions, because when a human might decide minutes or days later you persist the interruption to a durable store and rehydrate it to resume, which is the pattern the major SDKs formalize with serializable run state.

The tradeoff

A gate at the proxy means approval is enforced at the wire and is independent of the model's cooperation, so a model that ignores its instructions still cannot execute a gated write, which is the whole point, that you do not rely on the model's good behavior for safety-critical actions

and instead rely on the proxy.

> **Principle.** Pause the tool call on a promise at the interception seam, resolve it on the human's decision, and auto-deny on timeout. Make the approval card the consent surface, and enforce at the wire rather than in the prompt.

8 Streaming: make the agent legible

A multi-minute turn that shows a spinner is a broken product, so the job of the interface is to turn a black box into something a user can watch, trust, and interrupt.

Stream everything as typed parts

You model the agent's output as an ordered list of typed parts, the text, the reasoning, the tool calls, the tool results, and any custom data parts, and you stream each one incrementally over server-sent events while the interface re-renders the in-flight message on every chunk. Two practical notes apply, that you parse markdown incompletely and mid-stream so that partial tokens render as formatted text rather than raw markup flashing on screen, and that you send a heartbeat roughly every twenty-five seconds so the connection survives long silent thinking gaps, while setting buffering-off headers so that proxies do not swallow the stream, and the payoff is that perceived latency collapses, because the user sees activity within about a second rather than waiting for the whole turn.

Let tool calls render components

The frontier is generative interface, where the model's tool calls render structured interactive components, a chart, a table, an approval card, rather than a paragraph of markdown[14], on the principle that you do not make the model describe the weather when its tool call can render the weather card. The clean architecture goes from parts to blocks to typed renderers, where you group the flat parts stream into semantic blocks, have the agent emit rich interface through a single validated envelope with a stable identifier so that a later update replaces the same card in place, and dispatch each block to a lazy-loaded and individually fault-isolated renderer, so that one bad chart never crashes the message. You derive cards from tool results rather than from prose, and if you let the model emit executable interface you run it in a sandboxed iframe with no network access and keep it behind a flag.

Persist after the stream, resume by state

You return durable identity in the first byte, sending the conversation identifier as a response header immediately so that the URL is shareable and refresh-safe before the turn even finishes, and you persist once the stream drains so that the response is never blocked on a database write, while knowing the failure mode, that if the client disconnects mid-turn the callback

may not fire, so when mid-run survival matters you drive persistence from a server-side consumer of the stream that is independent of the client connection[15], since a refresh is a disconnect rather than a cancel. On reload you rehydrate from the stored parts, and the same renderer serves both live streaming and replayed history, so a refresh simply works.

The prompt-cache discipline streaming forces

You split the prompt into a stable cacheable prefix, the persona and system instructions that are byte-for-byte identical every turn, and a volatile per-turn block that holds everything specific to the user or the turn, because prompt caching rewards a stable prefix and nothing else, with documented reductions of up to roughly ninety percent of input cost and eighty-five percent of latency on the cached prefix[16][17]. You put static content first and variable content last, and on a multi-turn loop with a large system prompt this is a five to tenfold reduction in input cost for no quality loss, which is the single highest-leverage cost lever you have.

> **Principle.** Stream everything as typed parts, render the reasoning and every tool call legibly, let tool calls render components rather than paragraphs, persist after the stream and resume from stored state, and split the prompt into a cacheable prefix and a volatile tail.

9 Observability: trace every turn, distrust the cost dashboard

Trace every turn

Every turn should produce a trace, a parent span with child spans for each model call and each tool execution, which is standardizing as the semantic conventions for generative-AI telemetry[18], and per turn you capture a stable identity, the user and session identifiers, so that you can slice quality by cohort, the model, the token usage, and the latency, the source and a few quality-signal tags such as a refusal marker or an empty-completion flag, and a deterministic trace identifier derived from the message so that a retry upserts the same trace rather than fanning out duplicates. Two rules are hard-won, the first that if you swap or add a model runtime you verify trace coverage empirically, because a new harness that does not emit the spans your exporter expects goes silently invisible and you discover the hole only by querying for the new traffic and finding no rows, and the second that telemetry is never load-bearing on the request path, since every trace write, flush, and tag is best-effort and swallows its own errors, degrading to a no-op rather than a failed user turn.

Distrust the cost dashboard

A cost dashboard derived from a token count multiplied by a price table is an estimate rather than an invoice, and in one set of measurements a popular observability tool over-reported dollar cost against the actual cloud bill by between one and a quarter and twenty-four times depending on cache friendliness, while the token counts and the behavior were reliable and

only the dollar figures were inflated, for reasons the tool's own documentation confirms, that it cannot see reasoning tokens, does not count built-in tool costs, and only recently modeled tiered pricing[19]. You therefore reconcile against the cloud provider's actual billing before quoting any cost number.

Two health checks with opposite tolerances

You split health into two endpoints with opposite tolerances, a shallow check on the load-balancer hot path that probes the core dependencies but wraps them in a recent-success cache and reports healthy when any probe succeeded within a short window, which absorbs sub-second blips, and a deep check for operations and alerting only that runs the same probes unwrapped together with the full dependency set, giving strict live truth so that real failures still alarm. The lesson behind this design is a genuine postmortem, where a single shared dependency probed without tolerance on the hot path took down a whole fleet, because when one host froze every shallow check failed within a second, the load balancer removed all backends, and a single-host blip became roughly thirty-one minutes of total outage, which is why a shared singleton probed without tolerance is a fleet-wide single point of failure.

Log usage, gate payloads behind a privacy review

You log operational metadata on every tool call as fire-and-forget, the account, the tool, the latency, the success, and a stable error code that you index so that failures are queryable without a trace lookup, but the payloads, the raw arguments and the result fingerprints, carry personal and business-sensitive data, so you land those columns in the schema dark, nullable and reviewable, and gate the actual capture behind a single feature flag that you flip only after a documented privacy review rather than a code review.

> **Principle.** Trace every turn and verify coverage when you change runtimes, trust token counts and behavior while reconciling dollars against real billing, split health into a lenient check and a strict one, and log metadata always while gating payloads behind a privacy review.

10 Evaluation: testing something non-deterministic

You cannot snapshot-test a model's prose, so the discipline is to layer your checks by how deterministic each surface is, and to make the stochastic part testable by constraining it.

Layer one: constrain a stochastic surface into a contract, then unit-test the contract

Wherever you can, you force the model's behavior through a deterministic seam and test the seam rather than the prose, so you assert that a tool's argument schema rejects bad enumerations, lowercased identifiers, and empty arrays, you mock the network and assert that the tool

hits the right endpoint with the right body, you have the model emit a fixed marker line on a refusal and assert that the marker is detected at the start of the message rather than mid-message, and you assert that the tool description still contains the steering sentence you rely on, which is how a fix made in prose earns a regression test. The best practice is to encode the literal failing input harvested from production into the regression test so that it cannot drift from the real failure.

Layer two: a model graded by a model, where determinism is impossible

For free-form quality you use a model to grade a model, but you do it carefully[20], at low temperature, with structured output and a fixed rubric, scoring pass or fail rather than on a one-to-five scale, since a three against a four means nothing and a binary verdict forces clarity, using a different model family from the one being judged to remove self-preference bias, caching by input hash so that re-runs are deterministic, and keeping the grader informational rather than a hard merge-blocker, so that a stochastic score lands in a pull-request comment and a regression alert rather than a red cross, calibrated against a single domain expert.

Layer three: grade outcomes and trajectories, and probe live upstreams

The strongest primary source on agent evaluation makes the central point, that you grade the outcome rather than the path[21], because a booking agent might say that the flight is booked while the outcome that matters is whether a reservation exists in the database, and frontier models find loopholes that break brittle assertions about the exact sequence of tool calls, so you check the state of the world rather than the steps. You hold a golden-dataset bar, writing tasks where two domain experts independently reach the same verdict, and you start with twenty to fifty real failures rather than a perfect corpus. When correctness depends on a nondeterministic or undocumented upstream, a third-party interface whose real behavior you cannot assume from its specification, the source of truth is an empirical probe against a real account rather than a unit test, and you lock the findings as shape, differential, and negative-control assertions, that a narrowed query returns fewer rows than the baseline and that a bogus identifier returns none, which survive changing data while still catching a regressed wire shape, and you keep these out of the default test run, execute them manually before a merge, probe writes in dry-run mode, and make them fail loudly rather than skip silently.

> **Principle.** Constrain the stochastic surface into a contract and unit-test the contract, judge the rest with a low-temperature cross-family grader that is informational rather than blocking, grade outcomes over paths, and probe undocumented upstreams live while locking the shape.

11 Release safety: ship dark

The mental model

Agent features are risky and they iterate quickly, and the one rule that makes continuous shipping safe is that anything not ready for production users must be dark in production, so a code feature hides behind a registered feature flag that defaults to off, a whole service hides behind an infrastructure gate such as a compose profile or omission from the deploy loop, and a feature that has both a service and the interface that drives it gates both. The payoff is structural, because when all in-flight work is dark behind flags that default to off, promoting code from the integration branch to production is safe at any time, since there is no release window when nothing un-vetted is reachable.

What it means in practice

You keep one typed registry as the single source of truth, where each flag has a canonical name, a default of false, an owner, and a description, and you read it at call time through one helper rather than inlining an environment check, because reading at call time means an environment flip and a restart take effect with no redeploy, which is your instant rollback lever. You give a canonical value precedence over any legacy alias when you migrate an existing flag into the registry, which makes the migration a provable no-op, and you keep the registry narrow, holding simple booleans, because flags with bespoke semantics or build-time inlining need their own handling and folding them in naively would silently flip behavior.

> **Principle.** Keep flags that default to off in a typed registry, read them at call time, and gate both the code and the service. Dark by default decouples merge from release and turns rollback into an environment flip.

12 Multi-agent: when to fan out

The mental model

There is a real and well-documented disagreement here, and the resolution is a useful rule. The case against multiple agents is that single-threaded agents are more reliable because context stays continuous, since actions carry implicit decisions and conflicting decisions carry bad results, so parallel sub-agents that each make decisions produce inconsistent output[22]. The case for them is that for breadth-first reading and research tasks that exceed one context window and touch many tools, a lead agent with sub-agents outperformed a single agent by ninety percent on an internal research evaluation, at roughly fifteen times the token cost, which is worth paying only when the value of the task is high[3].

The synthesis

A later follow-up reconciles the two, holding that multiple agents can contribute intelligence in parallel while writes and decisions stay single-threaded[23], so the practical shape is to map, reduce, and manage, where a manager decomposes the work, isolated workers each take a read-heavy slice with clean context, and the manager serializes the writes. In a harness this falls out naturally, because the orchestrator, which is the model, decides when to fan out, configured by a hint in the system prompt rather than by hardcoded branching, and the workers are leaf agents with the same tool surface and a bounded depth, so the user sees a single agent reasoning about the problem while the delegation machinery stays internal and is never named in the user-facing reasoning.

> **Principle.** Fan out for breadth-first, read-heavy, parallelizable work, and only when the value of the task justifies roughly fifteen times the tokens. Keep writes and final decisions single-threaded, and let the model decide when to delegate.

13 Reference architecture

The pieces compose into a single production-grade system, where each layer is one more application of the boundary between what the model decides and what code guarantees.

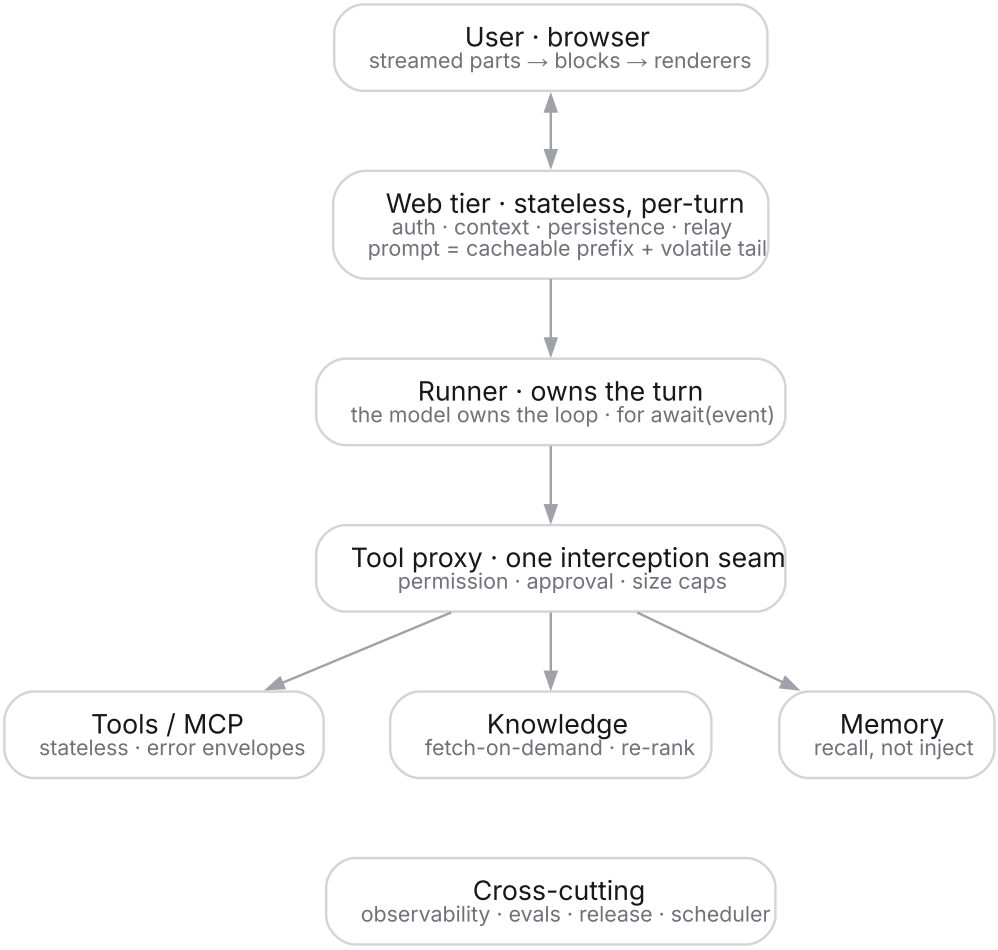

Figure 5. A production-grade single-agent system, end to end. The web tier assembles context and relays, the runner owns the turn while the model owns the loop, the tool proxy is the single interception seam, and tools, knowledge, and memory sit behind it, with observability, evaluation, release, and scheduling cutting across the whole.

14 If you remember nothing else

1. **Max prompt, min code.** Behavior lives in prose, and code is reserved for the irreversible, the financial, the cross-tenant, and the trust-fabricating, with scaffolding deleted as the models improve.
2. **Start simple.** A workflow comes before an agent and a single call comes before a workflow, with autonomy added only where the problem is genuinely open-ended.
3. **The harness owns the turn and the model owns the loop.** There is one interception seam between the model and the world, and you resume by thread handle rather than replay.
4. **Tools are contracts.** The descriptions and schemas are prompt surfaces worth investing in, errors come back in a form the model can recover from, and steering travels with the tool.
5. **Retrieval and memory are tools rather than prefixes.** You inject only what is small, per-turn, and always relevant, and fetch everything large or situational, with citations verified in code.

6. **Human approval pauses the tool call rather than the agent.** Approval is enforced at the wire, a timeout auto-denies, and the card is the consent surface.
7. **Stream everything and make it legible.** Typed parts, generative interface, visible tool calls, and a cacheable prefix with a volatile tail.
8. **Trace every turn and distrust the cost dashboard.** Tokens and behavior are the truth while dollars are an estimate to reconcile against the real bill.
9. **Evaluate in layers.** Deterministic contracts, an informational cross-family judge, and live upstream probes, grading outcomes rather than paths.
10. **Ship dark.** Flags that default to off, read at call time, with rollback as an environment flip.

The throughline, stated once more, is to give control to the model, keep the code a thin and safe substrate, and spend the engineering effort on the prompt, the tools, and the guardrails, in that order.

15 References

[1] R. Sutton. The Bitter Lesson. 2019. incompleteideas.net/IncIdeas/BitterLesson.html

[2] L. Martin. The Bitter Lesson, applied to agents (quoting H. W. Chung). 2025. rlancemartin.github.io

[3] Anthropic. How we built our multi-agent research system. 2025. anthropic.com/engineering/multi-agent-research-system

[4] Anthropic. Effective context engineering for AI agents. 2025. anthropic.com/engineering

[5] Anthropic. Building Effective Agents. 2024. anthropic.com/engineering/building-effective-agents

[6] OpenAI. GPT-5 Codex prompting guide, the agent loop and harness. 2025. cookbook.openai.com

[7] Anthropic. Effective harnesses for long-running agents. 2026. anthropic.com/engineering

[8] Anthropic. Writing Effective Tools for AI Agents. 2025. anthropic.com/engineering/writing-tools-for-agents

[9] Anthropic. Introducing the Model Context Protocol. 2024. modelcontextprotocol.io

[10] A. Singh et al. Agentic RAG: A Survey. arXiv:2501.09136. 2025. arxiv.org/abs/2501.09136

[11] Anthropic. Contextual Retrieval. 2024. anthropic.com/news/contextual-retrieval

[12] OpenAI. Agents SDK, human in the loop. 2025. openai.github.io/openai-agents-python

[13] LangChain. LangGraph, human in the loop. 2025. langchain-ai.github.io/langgraph

[14] Vercel. AI SDK, generative user interfaces. 2025. sdk.vercel.ai/docs

[15] Vercel. AI SDK, resumable streams. 2025. sdk.vercel.ai/docs

[16] Anthropic. Prompt caching. 2024. anthropic.com/news/prompt-caching

[17] OpenAI. Prompt caching. 2024. platform.openai.com/docs/guides/prompt-caching

[18] OpenTelemetry. Semantic conventions for generative AI. 2026. opentelemetry.io/docs/specs/semconv/gen-ai

[19] Langfuse. Model usage and cost, documented caveats. 2026. langfuse.com/docs/model-usage-and-cost

[20] H. Husain. Creating an LLM-as-a-Judge that drives business results. 2024. hamel.dev/blog/posts/llm-judge

[21] Anthropic. Demystifying Evals for AI Agents. 2025. anthropic.com/engineering

[22] Cognition. Don't Build Multi-Agents. 2025. cognition.ai/blog/dont-build-multi-agents

[23] Cognition. Multi-Agents, What's Actually Working. 2026. cognition.ai/blog

---

Where this came from

X-Arc is an applied AI research lab that builds and deploys agents for the companies it works with. The patterns in this guide are the ones we run in our own production systems, written down in vendor-neutral form so that they are useful to anyone building an agent product rather than tied to a particular stack, and nothing in here is proprietary. If something on these pages is relevant to work you are running, the contact form is on the site and we come back within two working days.