AGENT ENGINEERING

Building Effective AI Agents

Reliability, not capability, is the binding constraint on a production agent. An analysis of the prompt–code boundary, the line between what a model should be trusted to decide and what code must guarantee, traced through the subsystems of a working system.

Behavior on the steerable surface; code for the invariants.

Author X-Arc
Date June 2026

ABSTRACT

Language models have become markedly more capable at multi-step work, yet the systems built on them fail in production far more often than their performance in isolation predicts, and the failure is not one of capability. A demonstration measures a model once, attended, on a short chain of steps; a deployed agent is measured repeatedly, unattended, and across long chains, and under those conditions the property that decides whether the system works is no longer how good the model is but whether the system around it returns the same correct result on the two-hundredth run as on the first. We take that property, the reliability of the system rather than the capability of the model, to be the central engineering problem of agent design, and we argue that it cannot be drawn from the model, which is the source of the variance rather than its cure, and must instead be built into the layer the model cannot see. The organizing principle we derive is the prompt–code boundary: every outcome an agent produces is partitioned by how much variance it can tolerate, with the irreversible, the financial, the cross-tenant, and the trust-fabricating guaranteed in code and fail-closed, and everything recoverable expressed in prose and delegated to the model. The report derives this boundary from three measured properties of capable models and then traces it through the subsystems of a production agent, the harness, tools, retrieval, memory, human approval, streaming, observability, evaluation, release, and delegation, treating each as one more place the boundary is drawn. Every pattern described has been run in production and none of the specifics are proprietary.

1 Reliability is built around the model

A language model that performs well in a demonstration will often behave differently once it is deployed and left to run on its own, and the change is not a loss of capability. The conditions changed. A demonstration is one attended run on a short chain of steps, while production is many unattended runs on long chains, and across that shift the property that determines whether the system works is no longer how good the model is in isolation but whether the surrounding system produces the same correct result every time. Capability and reliability are different axes, and it is reliability that ships.

Reliability of this kind cannot be obtained from the model, because the model is where the variance originates. A larger model raises the height of the task an agent can attempt and leaves the run-to-run variance roughly where it was, so reliability has to be engineered into the layer the model cannot see, which is the code and the prompt that surround it. The design question is how that layer should be divided between the

two.

The division we have converged on, across the agents we run in production, expresses as much of the system's behavior as possible in the language the model reads, and reserves code for the small set of outcomes that can tolerate no variance at all. We call the line between the two the prompt–code boundary, and where it falls is a conclusion rather than a preference, following from three measured properties of capable models.

The first is that general methods which scale with computation outperform handcrafted ones by a large margin, the regularity Sutton named the Bitter Lesson[1]. A prompt is a general control surface that improves on its own as the model improves, whereas control flow written in code is handcrafted knowledge that the next model outgrows, so behavior placed in the prompt compounds with model progress while behavior placed in code becomes a ceiling[2].

The second is that, in a non-deterministic system, code is the higher-variance control surface, which inverts the usual intuition. A small change to control flow can cascade into large and hard-to-predict changes in behavior, an effect reported directly by teams that have tried to steer complex agents through code[3], whereas a change expressed in prose is a change to a single inspectable artifact that can be read and compared. The belief that code is the deterministic surface and prose the unpredictable one is exactly reversed once the quantity of interest is the behavior of an agent rather than the execution of a program.

The third is that attention is a finite resource. The model's context is a budget rather than a container, and the task is to curate the smallest set of high-signal tokens that produce the intended outcome rather than to supply ever more instruction[4], so the surface on which behavior is expressed must itself be economical.

Delegating behavior to the prompt is frequently read as granting the model unbounded discretion, and the code side of the boundary is what forecloses that reading. Reserving code does not mean removing it; it means placing it in exactly one position, around the outcomes that can tolerate no variance. Every decision the system makes is partitioned by a single question, how much variance the outcome can tolerate. Outcomes that can tolerate none, the irreversible, the financial, the cross-tenant, and those capable of fabricating trust, are guaranteed in code and fail-closed, with the model not trusted to produce them; outcomes that are advisory, interpretive, or recoverable are delegated to the model and steered by prose. The result is bounded autonomy rather than free will: the model is free only over the region where a wrong answer is survivable, and code holds the line over the region where it is not.

The model decides	Code guarantees
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

Table 1: What the model is trusted to decide, and what code must guarantee.

The signature of a system built this way is visible in its source. In one production agent the instruction document is roughly twelve hundred lines of structured prose while the code that assembles and dispatches it is a handful of small functions, and the behaviors a conventional system would encode in control flow, which tool to call and in what order, how to reconcile conflicting sources, what to assume when the user is silent, how to recover from a timeout, are all expressed in language. A consequence of the same arrangement is that most behavior defects are repaired by editing prose rather than code: a wrong default is corrected by adding three sentences to a tool's description, together with a test asserting that the description still contains them, and no branch is introduced. The unit of a fix is often a paragraph.

2 Workflow and agent: the autonomy spectrum

Not every system that uses a model is an agent, and the distinction is sharp enough to state precisely. A workflow orchestrates the model and its tools along predefined code paths, so the sequence of steps is fixed in advance by the engineer; an agent is a system in which the model directs its own process and tool use, so the sequence is decided at run time by the model [5]. Between a single model call and a fully open-ended agent lies a spectrum of increasing autonomy, and a system's position on it is a design variable rather than a default.

Figure 1: The autonomy spectrum, from cheapest and most predictable to most capable and least predictable.

Each step along the spectrum trades predictability, latency, and cost for the capacity to handle work whose shape cannot be known in advance. The empirical observation behind the whole spectrum is that the implementations which succeed are not the most autonomous but the ones whose autonomy matches the task [5], so an agent is warranted only where the number of steps cannot be predicted and the path cannot be fixed. A single product is rarely at one position throughout: a deterministic intake and routing layer can absorb the predictable majority of requests and hand only the open-ended remainder to an agentic core, so that autonomy, the most expensive setting on the dial, is spent where the problem genuinely requires it and nowhere else.

3 The harness: loop ownership and decision ownership

The most consequential structural decision in an agent system concerns the loop that drives a turn. A turn is a loop: the system assembles a prompt, the model responds, the tools it requested are executed, their results are returned to it, and the loop repeats until the model signals that it is done [6], the same cycle Anthropic frames as gathering context, taking action, verifying the work, and repeating [7]. The decision is who owns which part of it.

The reliable arrangement separates loop ownership from decision ownership. The surrounding system 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 often, and when to stop. The only loop the application itself runs is the one that iterates the model's output events; it iterates what the model produces and does not direct what the model decides.

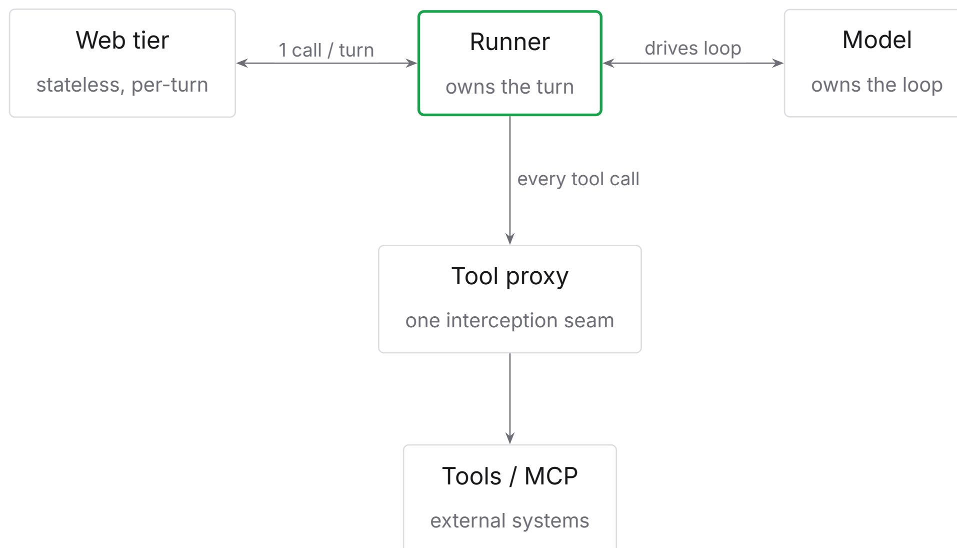

Figure 2: The runner owns the turn, the model owns the loop, and one proxy is the only seam to the world.

A second property follows from the first: there is exactly one seam at which the model meets the world. Every tool call is routed through a single proxy, and that proxy, rather than checks scattered through the codebase, is where permission, human approval, and response-size limits are enforced. Concentrating enforcement at one seam means the gate holds independently of the model's cooperation, because a denied call returns a structured error that is, from the model's vantage, indistinguishable from any other failure. It is the difference between a guarantee and a request.

Two further properties of the arrangement matter in practice. State is resumed by a thread handle rather than a transcript: because modern agent runtimes hold conversation state on the server, a turn is resumed by storing only the identifier of the thread and replaying nothing, which is dramatically cheaper than re-sending the history on every turn and is why an edit-and-resend feature is implemented as an explicit clearing of the thread rather than as a silent append. And one turn engine serves every origin: interactive chat, a scheduled run, and a channel integration share a single route discriminated by an origin field, which collapses the human-approval step where no human is present by mapping automatic approval onto the same permission machinery a person would otherwise drive.

The arrangement costs a network boundary between a stateless front end and a long-lived runner, and the cost buys three properties the monolithic alternative cannot provide: a front end that scales because it holds no state, a loop that survives a turn lasting minutes because it lives in a service built for it, and a security gate that is a single auditable seam rather than logic dispersed through a route that owns authentication, the model loop, tool execution, and persistence at once.

4 Tools as contracts

A tool is the only channel through which an agent perceives or changes anything beyond its own text, and the precise description is that a tool is a contract between a deterministic system and a non-deterministic one[8]. The contract has three surfaces, and the error most implementations make is to treat two of them as inert plumbing.

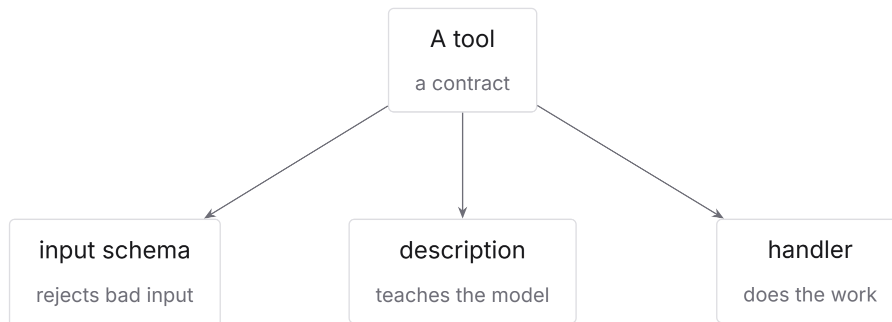

Figure 3: A tool has three surfaces; only the handler is code, the rest is prompt.

The schema is not only validation. It redirects, rejecting natural-language input and steering the model away from confusing an identifier with a display name. The description is not a label; it is a section of the prompt, and it teaches when the tool applies, when it does not, and the exact shape of its input. Only the handler is conventional code. Treating the description as prompt has a measurable consequence: production traces show that a vague description costs more tokens, through retry-after-empty loops, than a verbose one ever does, so the tokens spent on a careful description are recovered many times over.

Errors are returned as data the model can act on rather than thrown as stack traces. The envelope distinguishes fault classes, a client timeout from an upstream timeout from an internal error, so that the model and the dashboards attribute correctly, and it carries a recovery hint that is safety-aware for actions that are not idempotent.

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

Two integration properties recur. The Model Context Protocol is worth adopting as the transport[9] because it removes the need for a bespoke connector per model, and a tool server behind it is best built stateless and per-request so that authentication and tenant context cannot cross-contaminate between concurrent users. Steering must travel with the tool, because some hosts that connect over the protocol read only the tool list and never the system prompt, so the rules that must always hold are compiled

into the tool descriptions themselves and mirrored in the prompt, kept in lock-step, rather than stated once in a prompt a connector will never see. Tool exposure is then gated at three levels: registration decides which tools exist on a given turn by authentication scope and feature flag, the per-call permission is resolved at the proxy, and feature flags are read at call time. A convention makes the default safe: encoding the fact that a tool writes into its name, with a `propose_` prefix, reduces the default permission rule to a single line, that any unknown or `propose_` tool requires approval, which fails closed for any tool added later.

5 Retrieval as a tool, not a prefix

Retrieval is most often built as a prefix: the corpus is embedded, the matches nearest the query are selected, and the selected chunks are placed in the prompt ahead of the model. The arrangement makes the entire system hostage to a single retrieval call, because a mediocre selection cannot be repaired by a strong model that never sees the missing context and has no way to ask for it. The reliable arrangement treats retrieval as a tool the model invokes, on demand and repeatedly, deciding for itself when it has gathered enough[10], which is the same delegation the harness applies to control flow.

The decision of what to place in the prompt and what to leave behind a tool recurs across knowledge, skills, and memory, and it is the same decision each time. The rule that resolves it is that only what is small, specific to the turn, and always relevant is injected, and everything large or situational is fetched.

Concern	Where it lives	Why
Large knowledge corpus	`tool`	The relevant slice varies per question, so tokens are paid only when 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.
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.

Table 2: Inject or fetch: the recurring choice across knowledge, skills, and memory.

Several refinements follow from treating retrieval as a tool. The vector score is not the final ranking: a reliable pipeline over-fetches a wide candidate set and then re-ranks it deterministically, deduplicating near-identical content, weighting by source authority, and penalizing stale versions, so the model receives a small high-quality set rather than a large noisy one. Retrieval is a scaling tool rather than a default, and a knowledge base that fits within roughly two hundred thousand tokens is better placed whole in the prompt than retrieved[11]; when the corpus is large enough to require chunking, a chunk that reads only that revenue grew three percent has lost which company and which quarter, so a short chunk-specific context note prepended before indexing recovers a substantial fraction of otherwise-failed retrievals. Where grounded answers are the product, the prompt carries the policy, a hard rule that every claim be grounded in a tool result from the same turn, while the tool carries the content, never injected; a blanket version of that rule over-pulls, calling the knowledge tool even on operational turns that make no factual claim, so the rule gates advice rather than actions. Citations are the boundary in miniature: the model proposes a citation as an opaque identifier drawn from a tool result, and code resolves that identifier against what was actually retrieved, rendering a real reference when it matches and degrading to plain text when it does not, so the model decides what to cite while code guarantees the citation is real.

6 Memory as recall

The instinct is to build a memory store and prepend its contents to every prompt. The better arrangement is the one retrieval already established: the model is given tools to save, search, and read, and recalls on demand, while the prompt carries only a small content-free note announcing that the tools exist and when to use them. Recall is therefore tool-driven, which keeps memory out of the per-turn token budget and makes it explicit, inspectable, and deletable rather than passive background state.

The asymmetry between writing and reading is deliberate. A write is cheap and safe, so saving a salient fact can be proactive and ungated; a read is a judgment about relevance, so it stays at the model's discretion, with the single caveat, stated in the announcement, that the model search the whole store before concluding there is nothing, since without it models scope their search too narrowly and wrongly report an empty memory. Isolation is the one part that belongs unconditionally to code: memory is keyed by an identity derived on the server, a hash of the authenticated user under an issuer namespace, never an identifier the client supplies, because a store keyed by a client-set value is a cross-tenant leak waiting to happen, which is one of the guarantees on the code side of the boundary. The runtime gate that exposes the tools and the prompt gate that announces them are coupled so that the model is never told about a tool that is not wired, and a tool is never wired that the model is not told about.

7 Human approval at the wire

A consequential action, a write, a spend, or an irreversible change, requires a human gate, and the question is where the gate is placed. Placing it in the agent loop, as an instruction to ask before acting, makes safety contingent on the model's cooperation. The reliable arrangement pauses the tool call rather than the agent: at the interception seam the proxy waits on a promise that resolves only when a human decides, so that from the model's vantage the tool simply took longer and then returned either a result or a structured denial, and the loop never knows it stopped.

The invariant that the interrupt precedes the side effect is what the proxy enforces, and the canonical decision set, drawn from the major agent runtimes[12 13], is to approve, edit, reject, or respond. Several properties make the mechanism robust. The approval surface is a first-class interface element rather than a question in prose, so the consent is unambiguous and the agent's output stays clean. A timeout resolves as a denial returned through the same structured envelope, so a pending approval can never hang the turn; a real defect of this kind was a stale timeout constant set below the actual tool timeout, which silently denied valid actions, and the lesson was to keep the two values in sync and to comment the dependency. A decision can be remembered at two scopes, a per-tool, per-conversation always-allow that persists on the server and a session-wide automatic approval that is client state behind a one-time consent, and the paused state is serialized for the case where a human decides minutes or days later, so the interruption survives in a durable store and is rehydrated to resume. The whole arrangement exists so that a model which ignores its instructions still cannot execute a gated write, because the gate is at the wire and not in the prompt.

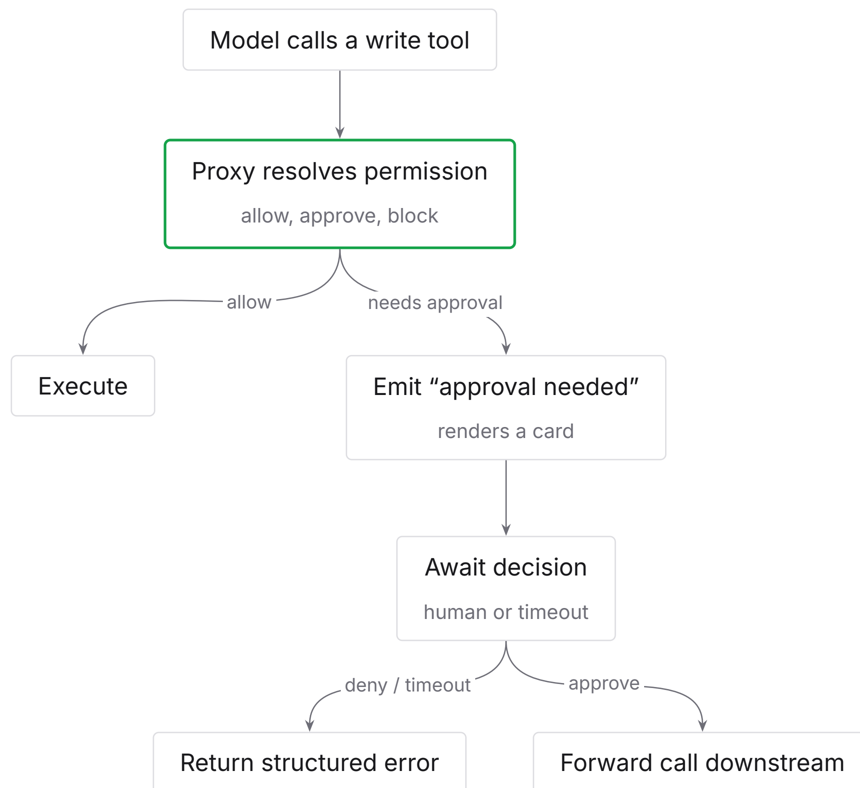

Figure 4: The tool call pauses on a promise at the proxy; the interrupt precedes the side effect.

8 Legible streaming

A turn that runs for minutes behind a spinner is a failure of the interface, not of the model, and the engineering task is to make a long, opaque process into one a person can watch, trust, and interrupt. The output is modeled as an ordered list of typed parts, text, reasoning, tool calls, tool results, and custom data, each streamed incrementally as it is produced, so the interface re-renders the in-flight message on every chunk. Two details make the stream survive contact with reality: markdown is parsed incompletely and mid-stream, so partial tokens render as text rather than as raw markup that flickers and resolves, and a heartbeat is sent on a short interval, with buffering disabled at the proxy, so a long silent reasoning gap does not look like a dropped connection.

The frontier of the interface is to let tool calls render structured components rather than describe them in prose, so a tool that retrieves a table renders the table and a tool that requests approval renders the approval card[14]. The clean form is a pipeline from parts to semantic blocks to typed renderers, where rich output is emitted through a single validated envelope carrying a stable identifier, so a later update replaces the same component in place, and each block is dispatched to a renderer that is loaded lazily and isolated so that one failed component cannot bring down the message.

Persistence and cost are the two properties the stream forces into view. Durable iden-

tity is returned in the first byte, so the conversation has a shareable, refresh-safe address before the turn finishes, and persistence is driven from a server-side consumer of the stream rather than a client callback, because a refresh is a disconnect and not a cancellation and the generation must outlive the connection [15]. The cost property is the prompt cache: the prompt is split into a stable prefix that is identical on every turn and a volatile tail that holds everything specific to the turn, because the cache rewards a stable prefix and nothing else, with documented reductions on the order of ninety percent of input cost on the cached portion [16 17], which on a multi-turn loop with a large system prompt is the single largest cost lever available and is obtained without any change to behavior.

9 Observability and the limits of measurement

An agent that cannot be observed cannot be operated, and each turn is made to produce a trace, a parent span with a child span for every model call and every tool execution, increasingly under the generative-AI telemetry conventions [18]. The trace carries the identity needed to slice quality by cohort, the model, the token usage, and the latency, a small set of quality signals such as a refusal or an empty completion, and a deterministic trace identifier derived from the message so that a retry updates the same trace rather than fanning out duplicates. Two properties are load-bearing. Trace coverage is verified empirically whenever a model runtime is swapped, because a new runtime that does not emit the expected spans goes silently invisible and the absence is discovered only by querying for traffic that returns no rows. And telemetry is never on the critical path: every write and flush is best-effort and swallows its own errors, degrading to a no-op rather than to a failed user turn.

One observation here is counter-intuitive enough to be worth isolating: the cost figure a tracing tool reports is an estimate, not an invoice. In our own measurements a widely used observability tool over-reported dollar cost against the actual cloud bill by between one and a quarter and twenty-four times, depending on how cache-friendly the workload was, while the token counts and the behavioral data it reported were sound. The discrepancy has documented causes, that the tool cannot see reasoning tokens, does not account for built-in tool costs, and only recently modeled tiered pricing [19], and the consequence is operational: a cost number is reconciled against the provider's billing before it is quoted, and a defensible figure is re-priced from captured token counts rather than read from the trace store.

A second result comes from a postmortem. Health was split into a shallow check on the load-balancer path and a deep check for alerting, which is correct, but the shallow check probed a shared dependency without tolerance, and when that single dependency froze every instance failed its check within a second, the balancer removed every backend, and a single-host stall became roughly thirty-one minutes of total

outage. A shared singleton probed without tolerance on the hot path is a fleet-wide single point of failure, and the correction is to wrap the probe in a recent-success cache so a sub-second stall is absorbed while a genuine failure still alarms.

10 Evaluation under non-determinism

A model's prose cannot be snapshot-tested, so evaluation is layered by how deterministic each surface is, and the stochastic part is made testable by constraining it. The first layer forces behavior through a deterministic seam and tests the seam rather than the prose: the argument schema is asserted to reject bad enumerations and empty arrays, the wire call is asserted against a mocked network to hit the right endpoint with the right body, a refusal is asserted to emit a fixed marker at the start of the message rather than mid-message, and a steering sentence the system relies on is asserted to be present in the tool description, which is how a fix made in prose acquires a regression test. The literal input that failed in production is encoded into the test so that it cannot drift from the real failure.

The second layer applies where determinism is impossible, grading free-form quality with a model [20]. The grader is run at low temperature with a fixed rubric and a binary verdict rather than a numeric scale, because a three against a four carries no information while a pass against a fail forces a decision; it uses a different model family from the one under test to remove self-preference, caches by input hash so re-runs are stable, and remains informational rather than a merge-blocker, since a stochastic score belongs in a comment and a regression alert and not in a red failure.

The third layer grades outcomes rather than paths [21]: a booking agent that reports a flight is booked is graded on whether a reservation exists, because a frontier model will find a path that satisfies a brittle assertion about the sequence of calls without achieving the result. The bar is a golden set on which two domain experts independently reach the same verdict, begun from twenty to fifty real failures rather than a perfect corpus. And where correctness depends on an upstream whose behavior cannot be assumed from its specification, the source of truth is an empirical probe against a real account rather than a unit test, with findings locked as shape and differential and negative-control assertions, that a narrowed query returns fewer rows than the baseline and a bogus identifier returns none, which survive changing data while still catching a regressed wire shape; such probes are kept out of the default suite, run before a merge, and made to fail loudly rather than skip in silence.

11 Release under uncertainty

Agent features are uncertain and iterate quickly, and the discipline that makes continuous shipping safe is that anything not ready for production users is dark in pro-

duction. A code feature is hidden behind a registered flag that defaults to off, a whole service is hidden behind an infrastructure gate, and a feature with both a service and an interface gates both. The property this buys is structural: when all in-flight work is dark, promotion from the integration branch to production is safe at any moment, because nothing unproven is reachable, and there is no release window to coordinate.

The flag registry is the single source of truth, each flag a canonical name with a default of false, an owner, and a description, and it is read at call time through one helper rather than inlined, because a call-time read means an environment change and a restart take effect without a redeploy, which is the instant rollback lever. A canonical value takes precedence over any legacy alias when an existing flag is migrated, which makes the migration a provable no-op, and the registry is kept narrow, holding simple booleans, because a flag with bespoke semantics or build-time inlining needs its own handling and would otherwise flip behavior silently.

12 Delegation and multi-agent systems

Whether to use more than one agent is a genuine and well-documented disagreement, and the resolution is a useful rule. The case against holds that a single-threaded agent is more reliable because its context stays continuous, since an action carries an implicit decision and parallel sub-agents that each decide produce conflicting and inconsistent output[22]. The case for holds that a breadth-first reading task which exceeds one context window and touches many tools is served better by a lead agent with sub-agents, which on one internal evaluation outperformed a single agent by ninety percent at roughly fifteen times the token cost[3], a price worth paying only where the value of the task is high.

A later synthesis reconciles the two: multiple agents may contribute intelligence in parallel, but writes and final decisions stay single-threaded[23]. 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 of the structure already in place, because the orchestrator is the model and decides when to fan out, configured by a hint in the prompt rather than by hardcoded branching, and the workers are leaf agents with the same tool surface and a bounded depth, so the delegation is internal and never named in the reasoning the user sees. It is the boundary once more: intelligence is delegated widely, while the writes that cannot tolerate conflict stay on a single thread.

13 Conclusion

The argument of this report is that the reliability of an agent is a property of the system around the model rather than of the model itself, and that the system is organized by

a single boundary, the one between prompt and code. Behavior is expressed in prose and delegated to the model wherever a wrong answer is recoverable, and expressed in code and withheld from the model wherever a wrong answer is not. Each subsystem examined here is that boundary drawn in a different place: the harness delegates the loop and owns the turn, the proxy delegates the call and owns the gate, retrieval and memory delegate selection and verify the result, human approval delegates the decision and holds the side effect, and release delegates iteration and keeps the unproven dark. The patterns are not independent techniques but the consequences of one decision applied consistently, which is why the difficulty of building an agent is concentrated in drawing the boundary well rather than in any single pattern. Control belongs to the model, code belongs to the small set of outcomes that must not vary, and engineering effort is best spent on the prompt, the tools, and the boundary between them, in that order.

14 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. rlance-martin.github.io

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

[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

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

Where this came from

X-Arc is an applied AI research lab. The patterns analyzed in this report are drawn from the agent systems we run in production, recorded in vendor-neutral form so that they transfer beyond any single stack, and none of the specifics are proprietary.

x-arc.ai