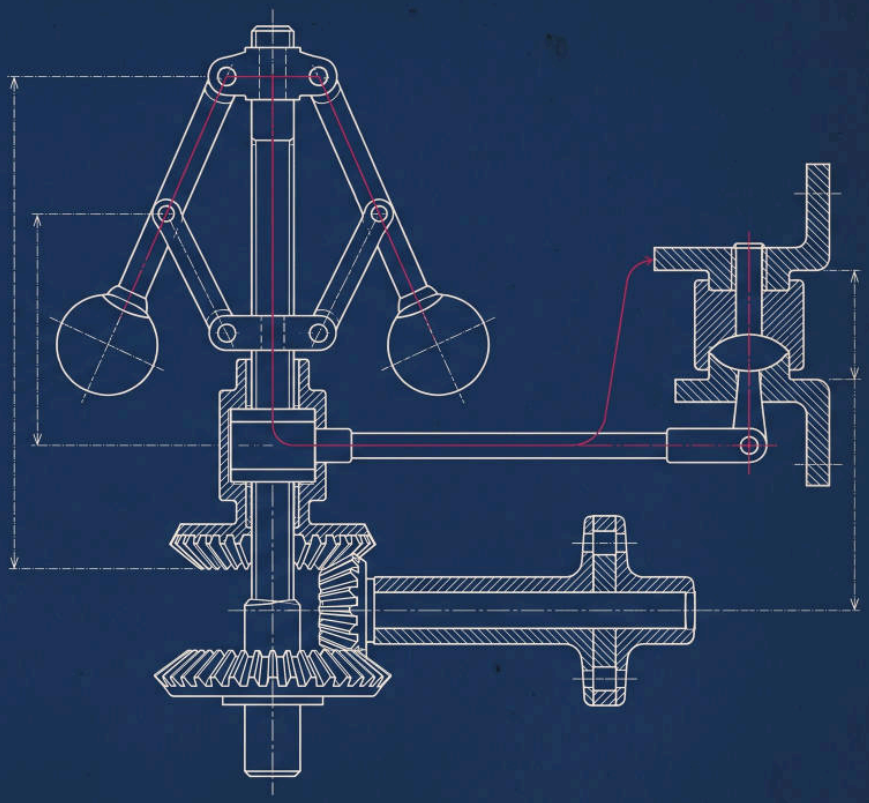

CHAPTER NINE

Agent Fundamentals

SECTIONS 9.1 TO 9.11

An agent is a model whose output determines what happens next. That sounds like a small change and it is not: it converts a bounded call into an unbounded process that decides for itself when to stop.

CHAPTER NINE

Contents

PART III · AGENTS

An agent is a model whose output determines what happens next. That sounds like a small change and it is not: it converts a bounded call into an unbounded process. Every dimension the previous eight chapters left open now gets exercised repeatedly, by something that decides for itself how many times.

9.1 What an Agent Actually Is

Strip the vocabulary away and an agent is a `while` loop containing a model call. That is not a dismissal; it is the correct level of abstraction, and it makes the engineering problems visible in a way that the word "autonomous" conceals.

> **The framing for all of Part III: an agent is a distributed system where one node is non-deterministic and bills you per message.** Every instinct you have about distributed systems applies, with one component that cannot be made reliable by retrying it, because retrying is what it already does.

Section 2.1 established that the model has no execution: it emits text that *looks like* a function call, and your code decides whether to act on it. That remains true in an agent, and it is the entire basis of the permission model in section 9.7. What changes is that the output now feeds back into the input, so an error at step three is present in the context at steps four through twelve, being reasoned about as though it were a fact.

The arithmetic that should govern your ambition

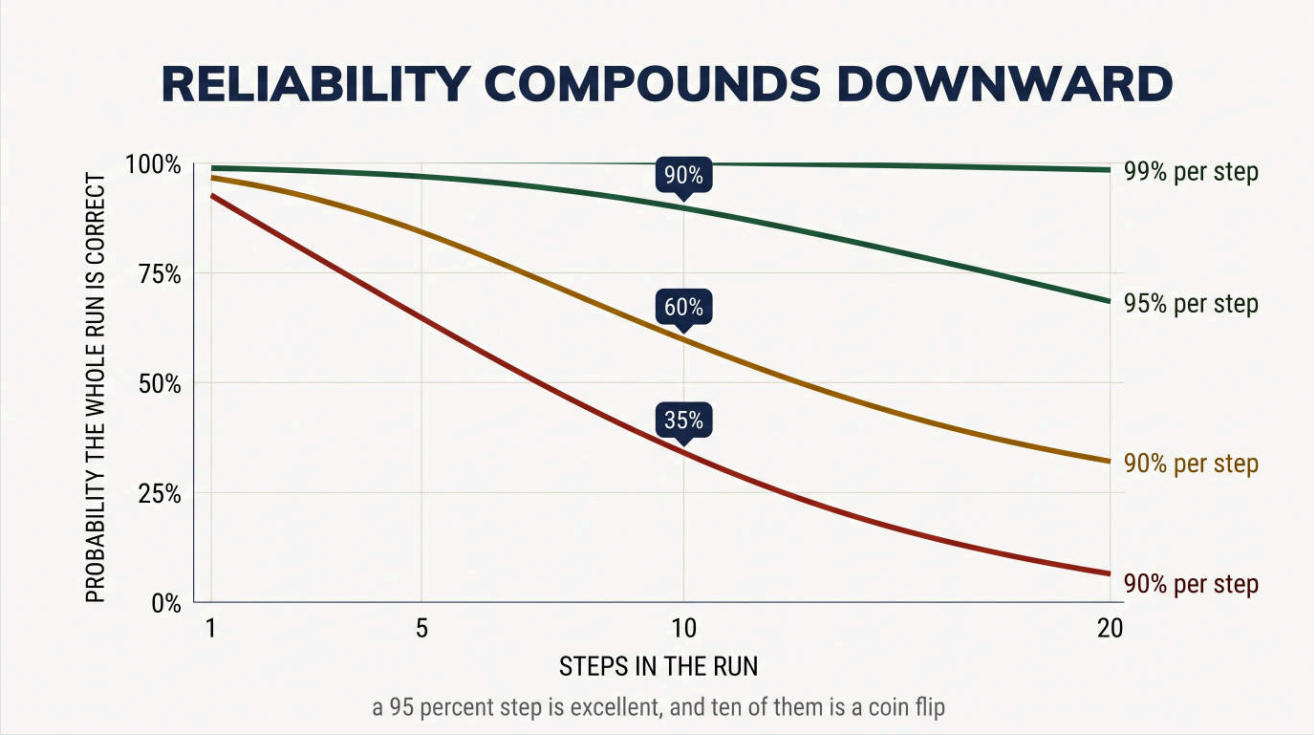

Figure 9.1 Green is 99 percent per step, amber 95, crimson 90. A 95 percent step is excellent in isolation, and ten of them is a coin flip.

```
def compound_reliability(per_step: float, steps: int) -> float:
    return per_step ** steps
```

Per-step accuracy	1 step	5 steps	10 steps	20 steps
99%	0.990	0.951	0.904	0.818
95%	0.950	0.774	0.599	0.358
90%	0.900	0.590	0.349	0.122

Read the 95 percent row carefully, because that is roughly where a well-engineered step sits after everything in Chapters 5 to 7. That placement is a field observation, not a law: measure your own per-step success rate from traces, which section 14.5's proxy signals give you without extra work, and read the row you actually sit in. Ten such steps in sequence completes correctly 59.9 percent of the time. Twenty steps is 35.8 percent, which is a system that fails more often than it succeeds while every individual component performs excellently.

```
def test_halving_the_steps_beats_improving_the_model():
    """Five steps at 95 percent beats ten steps at 97 percent."""
    assert compound_reliability(0.95, 5) > compound_reliability(0.97, 10)
    # 0.774 > 0.737
```

That test is the most useful line in the chapter. Shortening the loop beats improving the model, and improving the model is where almost everyone looks first. A two-point gain in per-step accuracy is a hard research problem; halving the step count is usually a design decision you can make on a whiteboard, by moving deterministic work out of the loop and into code.

> THE COROLLARY THAT DECIDES MOST AGENT DESIGNS
>
> If a step can be done deterministically, *do it deterministically*. Every step you remove from the loop multiplies reliability rather than adding to it. A pipeline of three fixed stages with one model call each is a different reliability class from an agent free to choose its own five steps, even when the total work is identical.
>
> Reach for an agent when the **sequence genuinely cannot be known in advance**. That condition is rarer than the enthusiasm for agents suggests, and Chapter 10's explicit graphs exist for the large middle ground where the sequence is mostly known.

9.2 The ReAct Loop and Termination

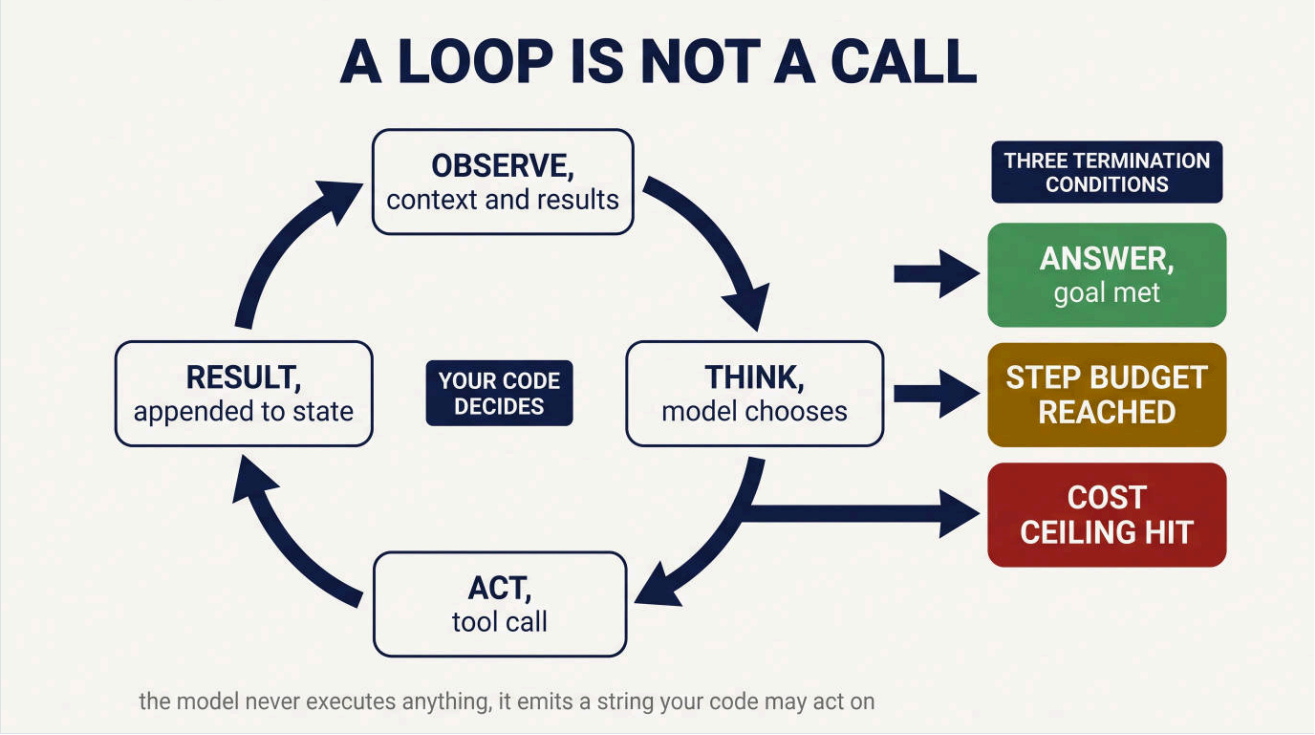

Figure 9.2 Observe, think, act, append. Six exits, and a loop with fewer is an outage waiting for the right input.

The loop has a name. **ReAct**, for reason and act: the model alternates a reasoning step with an action, and the result of the action, the observation, feeds the next reasoning step. It began as a prompting technique in a 2022 research paper, before tool calls were a native API surface; it survived because its shape matched what the APIs later made first class, and by now nearly every agent framework is a ReAct loop under its own branding. The name is worth keeping because a named pattern has nameable failure modes, and this chapter is largely a catalogue of them.

The pattern is four steps repeated: observe the current state, let the model choose an action, execute it if permitted, append the result. What matters architecturally is not the loop body but the exits.

```
# src/docassist/agents/loop.py
class Stop(str, Enum):
    ANSWERED = "answered"               # the goal was met
    STEP_BUDGET = "step_budget"         # too many iterations
    COST_CEILING = "cost_ceiling"       # too much money
    DEADLINE = "deadline"               # too much wall clock
    AWAITING_APPROVAL = "awaiting_approval"
    NO_PROGRESS = "no_progress"         # looping on the same call
```

Six ways to stop, and exactly one of them is success. That ratio is the honest shape of the problem: **the characteristic production failure of an agent is not a wrong answer, it is a run that never returns while billing you per iteration.** Chapter 10's graph engine declares its own version of this enum and adds members for stops only a graph can have, a failed node and a per-node loop guard among them; the vocabulary grows with the architecture, but the discipline of naming every exit does not change.

```
while True:
    # Check BEFORE the call, not after. Checking afterwards means you always
    # pay for one step past the ceiling.
    if (breach := budget.breach()) is not None:
        return RunResult(None, breach, tuple(steps), budget.cost_used)
```

Detecting a loop that is not progressing

```
sig = _signature(call)
seen[sig] = seen.get(sig, 0) + 1
if seen[sig] > self.no_progress_repeats:
    return RunResult(None, Stop.NO_PROGRESS, tuple(steps), budget.cost_used)
```

A model issuing an identical call for the third time is not reasoning, it is stuck. Detecting it costs a dictionary and saves the remainder of the budget, and in the repository's run it stops after two steps rather than fifty.

```
no progress    -> no_progress   steps= 2  cost=$0.1500  partial=True
step budget    -> step_budget   steps= 5  cost=$0.0005  partial=True
cost ceiling   -> cost_ceiling  steps= 2  cost=$0.6000  partial=True
```

One turn, several calls

The repository's loop carries one `ToolCall` per step, which keeps the code readable, but every 2026 API can emit several tool calls in a single assistant turn, and the loop must decide what that means. Two decisions, both yours rather than the model's. **Execution order and concurrency:** read-only calls can run concurrently, because their failure modes compose the way section 10.8 will describe for parallel branches; anything with a side effect runs serially, and every call, concurrent or not, passes through the same `authorise` chokepoint of section 9.7. A batch is not a licence to skip the check on call three because call one passed.

Partial failure: the protocol requires one result per call, so a batch where the second call timed out goes back as two observations, one result and one error, and the model sees exactly which failed. Collapsing the batch into a single "it failed" discards the information that would let the model retry only the broken half, which is the section 9.8 reflection rule paying for precise error messages.

Structurally the change is small: the `Step` record's single `call` generalises to a list of calls with a result each, and nothing else in the loop moves. The budget still charges per model turn, the no-progress signature covers the whole batch, and the exits are unchanged. Treat it as a representation detail, not a new architecture.

9.3 Tool Schemas as Contracts

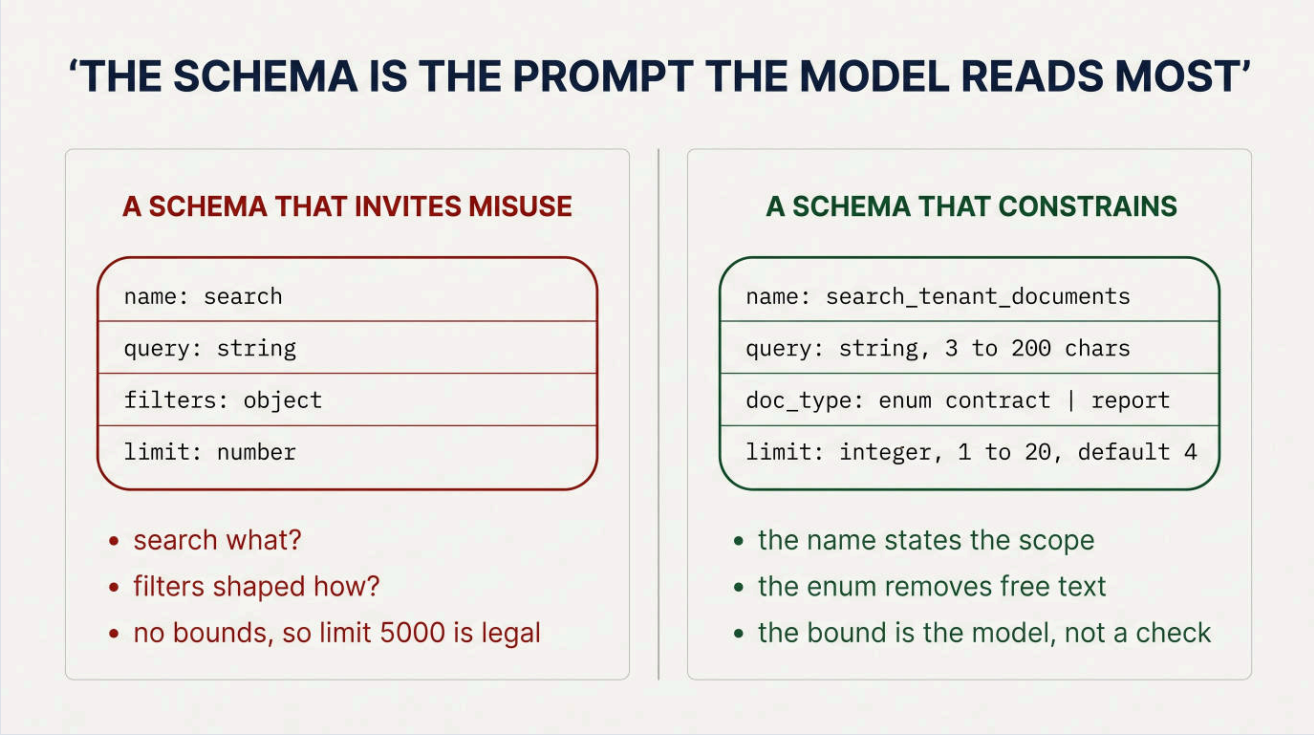

Figure 9.3 The schema is sent on every step of every run, which makes a vague description both a recurring cost and the cause of the failure it describes.

Teams treat the tool schema as an interface definition and it is closer to a prompt. It is transmitted on every iteration, the model reads it more often than it reads your system message, and its wording determines whether tools are selected and called correctly.

The retransmission cost is softened by provider prompt caching: the schemas sit at the front of a prefix that is identical from step to step, which is exactly the shape prefix caches discount, provided your assembly keeps them stable rather than reordering them per step. That is a second cache layer beside the semantic cache of section 3.6, and it discounts the cost without touching the quality argument: the model still reads every schema on every step.

Three properties separate a schema that constrains from one that invites misuse.

The name states the scope. `search` tells the model nothing about what corpus, which tenant, or what it will get back. `search_tenant_documents` answers all three before the description is read.

Enums replace free text wherever the set is known.

```
def enum_of(desc: str, values: list[str]) -> dict:
    """Prefer an enum to free text wherever the set is known. It removes a
    whole class of argument error rather than validating it after the fact."""
    return {"type": "string", "description": desc, "enum": values}
```

Bounds live in the schema, not in a check after the call.

```
def bounded_int(desc: str, *, low: int, high: int, default: int) -> dict:
    """A bound in the SCHEMA is enforced by constrained decoding (5.3).
    A bound checked afterwards is a rejected step you paid for."""
    return {"type": "integer", "description": desc,
            "minimum": low, "maximum": high, "default": default}
```

That distinction is the section 5.3 argument applied to tools. A `limit` field typed as `number` makes `limit: 5000` a legal call the model may reasonably attempt; a field bounded 1 to 20 makes it unrepresentable. One is a validation error you pay a step to discover, the other never happens.

> DESCRIBE THE FAILURE MODES IN THE DESCRIPTION
>
> The most valuable sentence in a tool description is usually the one saying what the tool does *not* do, or when it returns nothing. "Returns an empty list if the tenant has no documents of this type; this is not an error" prevents a model from retrying a correct empty result three times, which is a step-budget failure caused entirely by a missing sentence.

9.4 Tool Selection and Failure Modes

Four failure modes recur, and three of them are schema problems wearing different clothes.

Failure	Root cause, usually	Fix
Wrong tool chosen	Two tools whose descriptions overlap	Merge them, or make the names disjoint
Right tool, wrong arguments	Unconstrained parameter types	Enums and bounds (9.3)
Tool called repeatedly	The result was empty and the description did not say that is normal	Document the empty case; detect repeats (9.2)
No tool called at all	The model answered from parametric memory instead of retrieving	An eval case, and often a stronger instruction to ground

Advertise only what this run may call

```
def schemas_for(self, policy: Policy) -> list[dict]:
    """Only advertise what this run may actually call.

    Sending schemas for tools the policy forbids wastes tokens on every step
    and invites the model to attempt calls that will be refused, which burns
    a step to learn something you already knew.
    """
    return [t.schema() for t in self._tools.values()
            if t.name in policy.allowed_tools and t.risk <= policy.max_risk]
```

This is both a cost control and a quality one. Fewer tools in context means fewer tokens per step multiplied by every step of every run, and it means a smaller space for the model to choose wrongly in. **The most reliable agents in production tend to have three to five tools, not twenty.** That range is an observation from the field rather than a measured constant; the number that should govern yours is the wrong-tool rate in your own traces as the toolset grows, which is a per-step signal section 14.5 already collects.

A refused call is an observation, not an abort

```
except ToolDenied as exc:
    # Feed the refusal BACK as an observation. The model can choose another
    # tool; killing the run wastes everything spent so far on a recoverable
    # mistake.
    steps.append(Step(len(steps), thought, call, None, cost, latency,
                      error=str(exc)))
    continue
```

```
async def test_a_denied_tool_is_fed_back_rather_than_killing_the_run():
    out = await AgentLoop(REGISTRY, planner).run("q", READ_ONLY, Budget())
    assert out.stop is Stop.ANSWERED
    assert out.steps[0].error and "not permitted" in out.steps[0].error
```

In that run the model attempted `send_email` under a read-only policy, was refused, chose search instead, and answered. Aborting would have discarded everything already spent to punish a recoverable mistake. The same argument applies to tool exceptions: a timeout or a malformed result is an observation the model can act on, and only the budget should end a run.

9.5 Agent State and Scratchpad Management

The scratchpad is the conversation the agent has with itself, and it grows on every step. Section 2.10 established that the window is a budget; here it is a budget being spent by something that decides for itself how fast to spend it.

Two costs compound, and they compound in different directions.

Tokens. Step ten's prompt contains steps one through nine, so cost per step rises as the run proceeds. A ten-step run does not cost ten times a one-step run; it costs considerably more, because the input grows monotonically. Provider prompt caching blunts this, because a scratchpad that only appends is a stable prefix growing at its end, which is the shape prefix caches discount; it sits beside the semantic cache of section 3.6 and it reduces the bill, not the dilution.

Dilution. Section 2.10's distractor effect applies to the agent's own history. A failed tool call from step two is still in context at step nine, being reasoned about. Long scratchpads make agents worse in exactly the way long retrieval contexts do: section 2.10's context rot, self-inflicted one step at a time.

Strategy	What it keeps	Right when
Full history	Everything	Runs under about five steps. Simplest and correct at that length.
Windowed	The goal, plus the last *n* steps	Long runs where recent state dominates. Cheap, and it can forget a fact from step two.
Structured state	A typed object the loop updates, plus recent steps	The best default past five steps. Findings live in a field, not in prose the model must re-read.
Summarised	A compacted narrative of earlier steps	Long-horizon work. Costs a model call and can lose the detail that mattered.

Prefer structured state to summarisation. A summary is lossy in ways you cannot predict and costs a call to produce. A typed `findings` object updated by your code after each step is lossless for the things you decided matter, free, and inspectable in a trace. Summarise only what does not fit a schema.

9.6 Step Budgets and Cost Ceilings

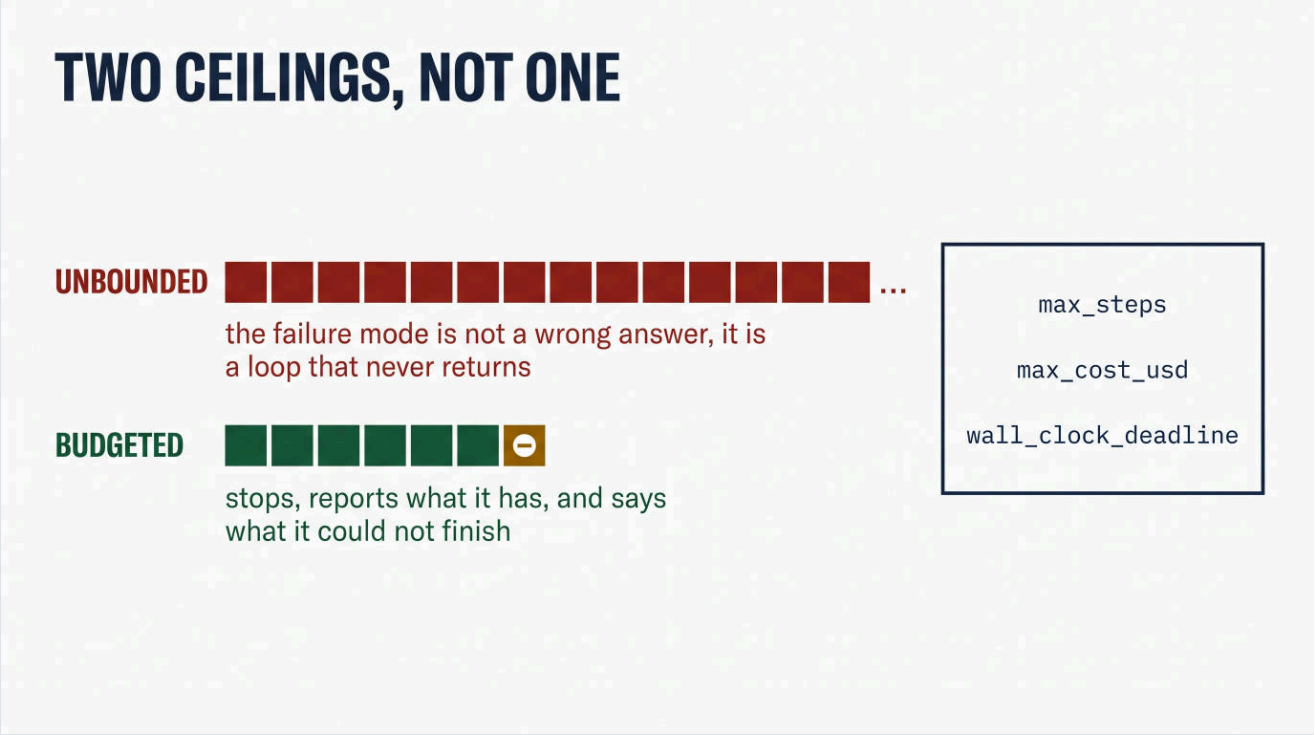

Figure 9.4 Three limits, because they bound three different failures.

```
@dataclass
class Budget:
    """Two ceilings and a deadline, because they bound different failures.

    max_steps catches a loop that iterates cheaply forever.
    max_cost_usd catches a loop that iterates expensively a few times.
    deadline_s catches a loop whose steps are individually slow.
    A run can breach any one without breaching the others.
    """

    max_steps: int = 8
    max_cost_usd: float = 0.50
    deadline_s: float = 120.0

    # The ceilings are configuration; these two are state. They record what
    # the run has spent, and chapter 10 checkpoints and restores them so a
    # resumed run cannot start with a fresh budget (10.5).
    steps_used: int = 0
    cost_used: float = 0.0
```

The repository's runs show why one ceiling is insufficient. A cheap loop burned **five steps for five hundredths of a cent** and was stopped by `max_steps`. An expensive loop breached the cost ceiling in **two steps**, nowhere near the step

limit. A single limit misses one of those entirely, and which one it misses depends on a model's choices rather than your design.

A budget stop is a partial result, not a failure

```
@property
def partial(self) -> bool:
    """A budget stop is not a failure. It is a partial result, and it must
    be reported as one rather than as an answer."""
    return not self.stop.is_complete
```

This matters more than it appears. An agent stopped at its ceiling has usually done real work: it has retrieved documents, established facts, narrowed the question. Discarding that and showing an error wastes what you paid for. Returning it *as though it were a complete answer* is worse, and is the agent version of section 6.9's confident wrong answer.

The correct behaviour is the same as the abstain path: report what was established, state what could not be finished, and make the incompleteness visible. "I found the notice period in the master agreement but could not check the amendment before reaching my limit" is useful. A confident summary of two-thirds of the evidence is not.

SET THE CEILINGS FROM YOUR OWN TRACES, THEN HALVE THEM

Teams set `max_steps` to twenty because it sounds safe. Section 9.1's arithmetic says a twenty-step run at 95 percent per step succeeds about a third of the time, so a ceiling of twenty is mostly a licence to spend money on runs that will not complete correctly anyway.

Look at the distribution of step counts for runs that *succeeded*. If the ninety-fifth percentile is four, a ceiling of six is generous and a ceiling of twenty is a budget leak with no quality upside.

9.7 The Permission Model

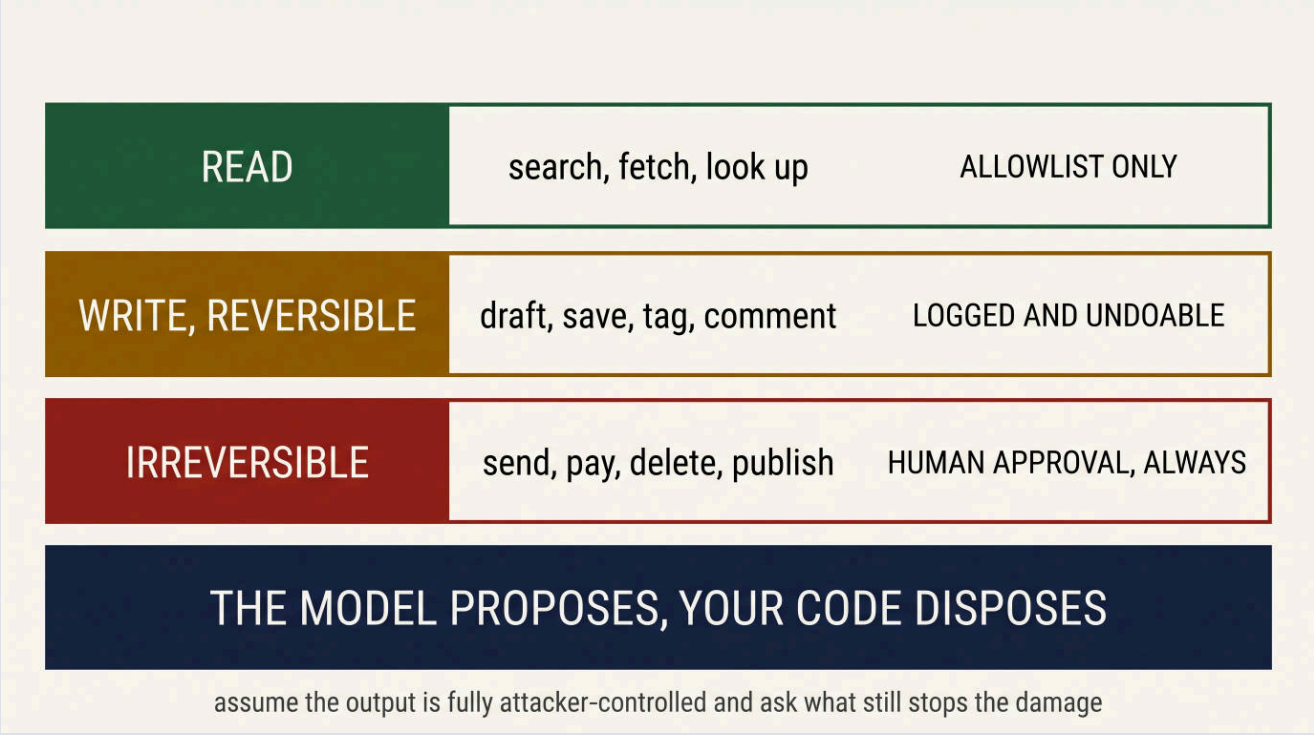

Figure 9.5 Tiered by what a wrong call costs, not by how hard the tool was to build.

This is where section 5.9's threat model becomes an enforcement mechanism, and where section 1.5's decision to keep credentials away from anything the model can reach starts paying.

The test, stated operationally: assume the model's output is fully attacker-controlled, then ask what still stops the damage. The answer must never be "the model would not do that". After section 5.9, you know an attacker who can place text in a retrieved document has a channel into the model's context, and after section 3.7, so does anyone who can put text in an image.

```
class Risk(IntEnum):
    """Ordered by what a wrong call costs, not by how hard it is to build."""
    READ = 0              # search, fetch, look up
    WRITE_REVERSIBLE = 1  # draft, tag, comment: logged and undoable
    IRREVERSIBLE = 2      # send, pay, delete, publish
```

One chokepoint, four checks, in order

```
def authorise(self, call: ToolCall, policy: Policy) -> Tool:
    """The single chokepoint. Order matters: existence, then allowlist,
    then risk ceiling, then approval."""
    tool = self.get(call.name)
    if tool is None:
        raise ToolDenied(f"unknown tool: {call.name}")
    if tool.name not in policy.allowed_tools:
        raise ToolDenied(f"{tool.name} not permitted for this run")
    if tool.risk > policy.max_risk:
        raise ToolDenied(f"{tool.name} exceeds the risk ceiling")
    if tool.risk is Risk.IRREVERSIBLE and call.name not in policy.approved_calls:
        raise ApprovalRequired(call)
    return tool
```

Three design decisions in ten lines.

Enforcement lives in the loop, never in the tool. A tool that checks its own permissions can be bypassed by adding a tool that does not, and someone will add one on a Thursday. The chokepoint is the property that makes the control auditable: there is exactly one place to read.

The allowlist and the risk ceiling are independent. A tool can be allowlisted and still refused for exceeding the ceiling, which is what lets one policy object serve a cautious tenant and a permissive one without editing tool lists.

```
def test_the_risk_ceiling_is_enforced_even_if_the_tool_is_allowlisted():
    """Allowlisted AND above the ceiling must still fail. Two independent
    controls, checked independently."""
    policy = Policy(tenant_id="acme", allowed_tools=frozenset({"send_email"}),
                    max_risk=Risk.WRITE_REVERSIBLE)
    with pytest.raises(ToolDenied):
        REGISTRY.authorise(ToolCall("send_email", {}, 0), policy)
```

Approval is per call, not a mode. A global "approved" flag means one approval authorises every subsequent irreversible action in the run, which is precisely the thing an injected instruction would exploit.

Approval suspends the run rather than blocking it

```
except ApprovalRequired:
    return RunResult(None, Stop.AWAITING_APPROVAL, tuple(steps),
                     budget.cost_used, pending_call=call)
```

The run returns with its state, its cost so far, and the pending call. A human sees exactly what is proposed, approves or rejects, and the run resumes from where it stopped. Chapter 8's result store and Chapter 10's checkpointing are what make that resumption possible; a run that must block a worker while waiting for a human is a worker held for an unbounded period, which is section 8.1 all over again.

> DESIGN THE APPROVAL SO IT CAN ACTUALLY BE READ
>
> An approval prompt reading "the agent wants to call send_email, approve?" trains people to click yes. Show the rendered effect: the recipient, the subject, the body, and the evidence the agent used to compose it. An approval gate that is not read is theatre with an audit trail.

9.8 Error Recovery and Reflection

Reflection means letting the model see its own failed attempt and try again. It works, and it has a specific cost profile that determines where it belongs.

Like section 9.2's ReAct, this is a named pattern, the **Reflection pattern** of the agentic canon: generate, critique, refine, with the critique usually written by a model, sometimes by a second model given a fresh context. Stated that way it sounds like a free quality upgrade, and framework documentation tends to present it as one. The whole question is where the critique comes from. A critic that shares the generator's context inherits the generator's blind spots, so the version this book endorses is narrower: reflect when an external system supplied the failure signal, and send judgements of correctness to critics that are genuinely outside, the grounding check of section 6.9, the verifier of

section 4.5, or the calibrated judge of section 14.4, which is the critic industrialised into evaluation. In a graph the same pattern appears as a draft-and-revise cycle, and section 10.9's loop guard is what keeps it from revising forever.

Situation	Reflect?	Why
Malformed tool arguments	Yes, once	The error message is concrete and the fix is usually mechanical.
Tool returned an error	Yes, once	The model can choose a different tool or different arguments.
Empty result	Only if the description says empty is meaningful	Otherwise the model retries a correct answer forever (9.4).
Answer failed a grounding check	Rarely	The same context produced it. Section 5.9's "ask the model to check itself" objection applies.
Budget breached	Never	Reflection costs a step you do not have.

The bounded-repair rule from section 5.8 transfers directly: **one retry, not a loop.** A model that cannot produce a valid call after seeing the error will not produce one on the seventh attempt, and you will have paid seven times to learn it.

> REFLECTION IS NOT VERIFICATION
>
> An agent asked to check its own work will frequently declare it correct, for the reason section 5.9 gave: the same compromised or mistaken context produces the check. Reflection is useful for *mechanical* failures where an external system supplied the error message. It is close to worthless for judging whether an answer is right, and that job belongs to the grounding check of section 6.9 or the verifier of section 4.5, both of which are external.

9.9 Multi-Agent Patterns and When Not To

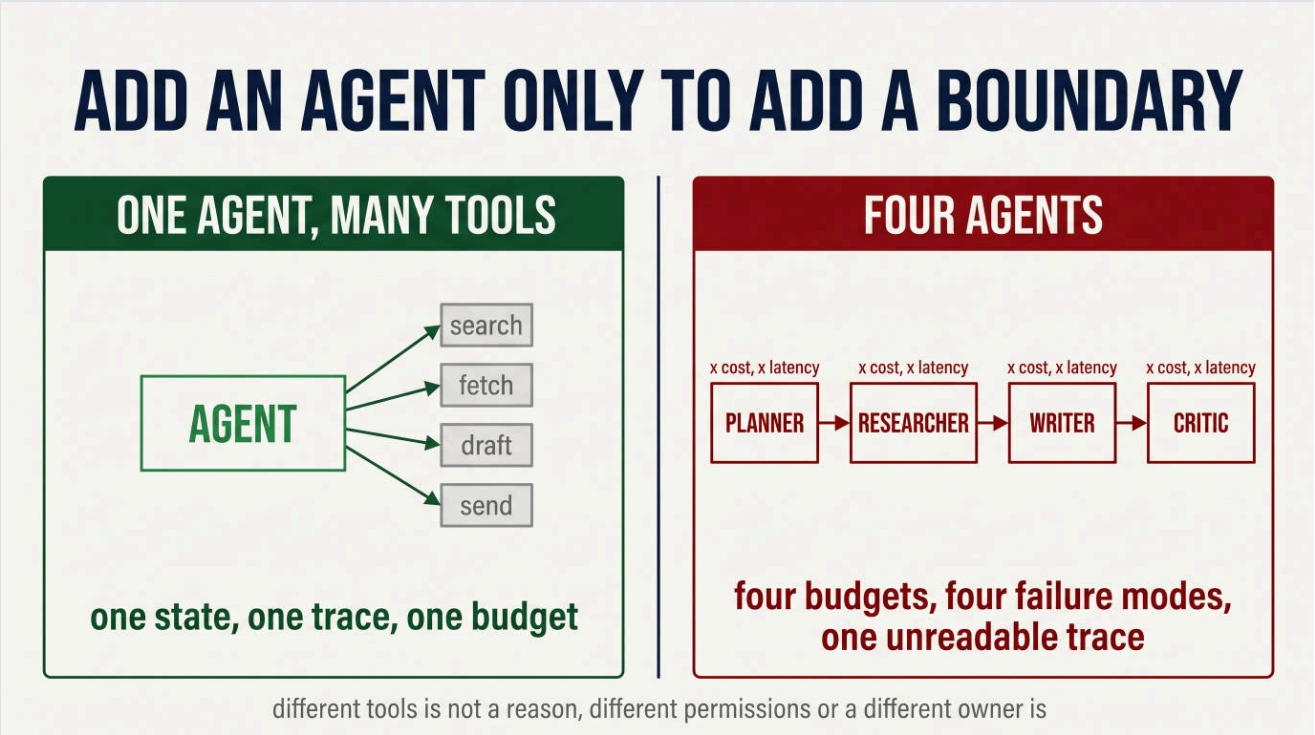

Figure 9.6 Four agents is four budgets, four failure modes and one trace nobody can read.

Multi-agent architectures are the most over-adopted pattern in this field, and the reason is that the diagrams are appealing while the costs are invisible until you are running one.

Section 9.1's arithmetic is the first argument against them. Four agents in a chain do not have four opportunities to be right; they have four opportunities to be wrong, and the reliabilities multiply exactly as steps do. A planner, researcher, writer and critic at 95 percent each complete correctly 81 percent of the time, before any of them has taken a step.

The costs that do not appear in the diagram:

- **Cost and latency multiply**, because each agent runs its own loop with its own scratchpad.

- **Four budgets**, and a global ceiling across them that someone has to design.
- **The trace becomes unreadable.** Section 9.10's per-step span is hard enough to reason about within one run; across four nested runs it is an archaeology exercise.
- **Information is lost at every handoff**, because each agent passes a summary rather than its state, which is section 9.5's summarisation problem occurring three more times.

The rule: add an agent only to add a boundary. Different tools is not a boundary; one agent can hold many tools. A *different permission set* is a boundary. A *different owner or team* is a boundary. A *different failure containment requirement* is a boundary. If you cannot name which boundary a second agent creates, you have added cost and latency to buy a diagram.

One more boundary is legitimate and easy to miss because it is about tokens rather than trust: **context isolation**. A search that must read forty files to answer one question fills the caller's scratchpad with material that will still be diluting it at step twelve, which is exactly section 9.5's problem. Delegating that search to a subordinate agent whose context is discarded when it returns means the caller receives a conclusion, not the transcript of finding it.

The test is the same shape as the others: name what the boundary protects. Here it is the caller's context budget, and the delegation pays for itself only when the intermediate material is large relative to the conclusion. Delegating a lookup that returns one line buys nothing and costs a second loop.

The pattern that does earn its keep is **a privileged agent calling an unprivileged one**: a read-only research agent whose output is treated as untrusted data by a supervising agent that holds the write permissions. That is a genuine security boundary, and it is the multi-agent shape Chapter 12 develops when third-party tools enter the picture.

The pattern zoo, translated

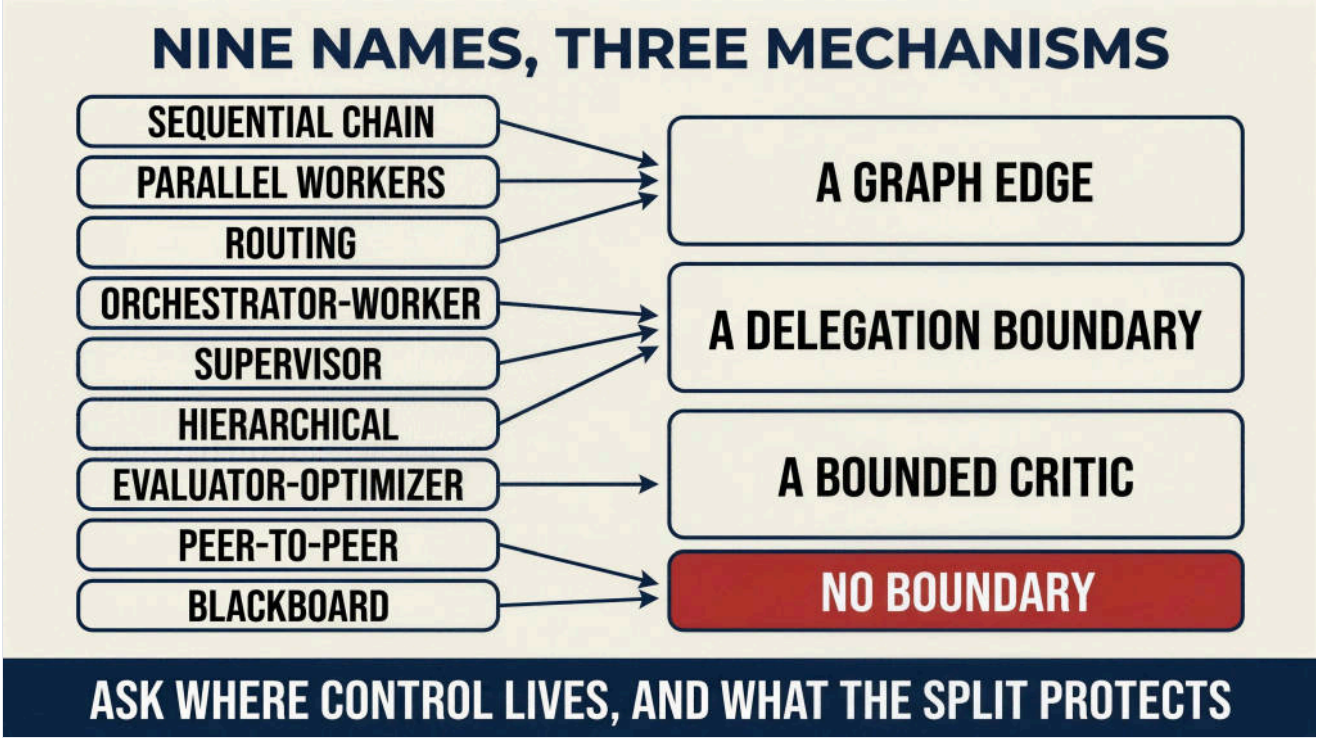

Figure 9.7 Nine names from the field, three mechanisms from this book, and two boxes with no boundary at all.

The field has settled on a vocabulary of agent workflow patterns, and you will meet the names in every framework's documentation and every design meeting: orchestrator-worker, supervisor, sequential chain, parallel workers, hierarchical, routing, evaluator-optimizer, peer-to-peer, blackboard. The names are useful shorthand. What they are not is nine different architectures, and translating them into this book's vocabulary shows which decisions each one is actually making.

Three of them are not multi-agent at all. A sequential chain is a pipeline: Chapter 10's graph with straight edges, and the stages want to be prompts, not agents, for section 9.1's arithmetic reasons. Parallel workers are section 10.8's fan-out, and the design content is the part the pattern name omits: the declared merge rule and the failure policy for a slow or dead leg. Routing is section 3.6's router, or in a graph, section 10.3's declared branch; the pattern diagrams draw a model choosing, and the first question is section 10.1's, whether a threshold could choose instead.

Three of them are one pattern at increasing depth. Orchestrator-worker, supervisor and hierarchical all describe delegation: one privileged loop that holds control flow and hands bounded subtasks to subordinates whose results it

treats as data. Whether that split earns its cost is exactly this section's rule, one nameable boundary per delegation: a permission set, an owner, a containment requirement, or the caller's context budget. What the hierarchy diagrams do not show is that section 9.1's arithmetic and section 9.5's handoff losses apply at every level, so depth needs a reason each time, not a habit.

One of them this chapter already built. Evaluator-optimizer is section 9.8's reflection with the critique promoted to a loop, and it inherits reflection's whole caveat: it works when the evaluator is genuinely outside the generator, a deterministic scorer or section 14.4's calibrated judge, and degrades into self-approval when it is the same context reading its own work. Bound the cycle the way section 10.9 bounds any cycle, or the optimiser will happily oscillate at your expense.

Two of them remove the property everything else here depends on. Peer-to-peer coordination and the blackboard share one design decision: mutable state and control flow with no chokepoint. There is no single authorisation path for section 9.7 to gate, no spine for section 9.10's trace to follow, and a blackboard every agent may write is section 11.9's poisoning surface with the write gate removed. The honest translation of these two is not an architecture but a question: which boundary does each peer add, and if the answer is none, the blackboard is four budgets and one incident report.

Use the names; they compress well. But interrogate every box in the diagram with the same two questions this chapter has been asking all along: where does control flow live, and what does this split protect. Most of the zoo collapses to a graph, one bounded agent, and a boundary you can name.

9.10 Observability for Agent Runs

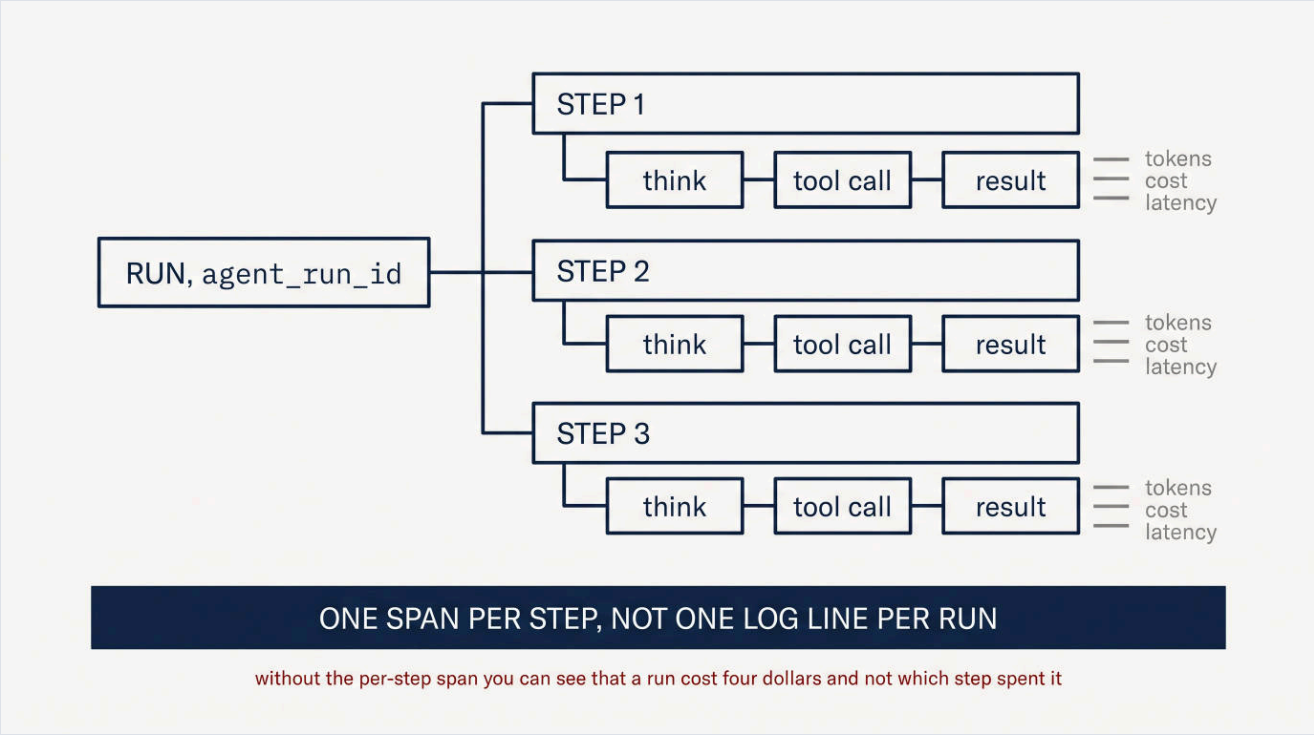

Figure 9.8 Without the per-step span you can see that a run cost four dollars and not which step spent it.

Chapter 1's trace seam recorded one span per model call. An agent run is many model calls with a causal relationship between them, so the trace must be a *tree* rather than a list.

```
@dataclass(frozen=True)
class Step:
    index: int
    thought: str
    call: ToolCall | None
    result: str | None
    cost_usd: float
    latency_ms: float
    error: str | None = None
```

Four questions arrive after every agent incident, and each needs a specific field to be answerable.

The question	What answers it
Why did this run cost four dollars?	Per-step `cost_usd`, summed and sorted
Where did it go wrong?	The first step whose `error` is set, or whose result contradicts a later claim
Why did it stop?	`Stop`, which distinguishes six outcomes rather than success and failure
What did it nearly do?	`pending_call` on an approval stop, and the `error` on every denied call

That last row is worth dwelling on. **Denied calls are a security signal, not noise.** A run that repeatedly attempts a tool outside its allowlist may be a confused model, and it may be an injected instruction from a retrieved document doing exactly what section 5.9 described. Aggregate denials by tool and by tenant, and alert on a rate change rather than on an individual event.

9.11 Three Agent Architectures Compared

GOOD

The unbounded loop

```
while True:
    action = llm(prompt + scratchpad)
    if action.done: return action.answer
    scratchpad += run_tool(action)     # any tool, no limit, no budget
```

Correct for a prototype, on a read-only toolset, run by someone watching it. It answers "does this task decompose at all", which is a real question worth an afternoon.

What it leaves unbounded: every dimension in the book at once. Steps, cost, wall clock, tool reach, and blast radius. The characteristic incident is not a bad answer; it is a run discovered the next morning having iterated for eleven hours.

BETTER

Bounded, allowlisted, traced

The loop of section 9.2 with the budget of 9.6, the registry of 9.3 and the permission chokepoint of 9.7. Runs terminate for one of six named reasons, every step is a span, and a denied call is an observation rather than an abort.

What it still does not solve. The agent chooses its own path, so two identical questions can take different routes and cost different amounts, which makes regression testing hard. Irreversible actions are gated but the gate is per run rather than per policy. And there is no checkpoint: a worker restart loses the run, which after Chapter 8 you know is not a hypothetical.

BEST

Constrained topology with durable state

The path becomes an explicit graph, the state becomes a typed object checkpointed after every step, and the loop becomes a durable job in the sense of Chapter 8. This is Chapter 10, and the summary of its argument is that **most of what people want from an agent is available from a graph with one or two model-chosen branches, at a fraction of the variance.**

Axis	Good: unbounded	Better: bounded	Best: constrained graph
Termination	Only on success	Six named conditions	Six, plus per-node limits
Cost per run	Unbounded	Ceilinged, high variance	Ceilinged, low variance
Blast radius	Every tool, always	Allowlist and risk tier	Per-node permissions
Recoverability	Run is lost on restart	Run is lost on restart	Resumes from checkpoint
Testability	Not meaningfully testable	Whole-run eval only	Per-node eval
Build cost	An afternoon	One to two weeks	Three to five weeks

A note on building versus adopting. Agent SDKs and hosted runtimes exist, they implement most of the Better tier out of the box, and adopting one is a reasonable decision. This chapter hand-builds the loop because the failure modes live in the loop's decisions, not in its plumbing, and you cannot evaluate a runtime you do not understand at that level.

The adoption test is therefore concrete: whatever you adopt must expose the same controls this chapter built. Named stop reasons, not a boolean. Step, cost and deadline ceilings you set. A single authorisation chokepoint your code owns, with approval as a suspension rather than a block. A runtime that hides any of those has not removed the requirement; it has moved you back to the Good tier with better branding.

> PROMOTION TRIGGERS
>
> **Good to better**, before any agent touches a tool with a side effect, or runs where nobody is watching. There is no defensible reason to delay: the budget is thirty lines and the permission chokepoint is ten.
>
> **Better to best**, when any of these hold: runs regularly exceed five steps; cost variance between identical requests is material; a run must survive a deploy; or you need to evaluate part of the pipeline rather than only the whole. *Most teams should get here faster than they expect*, because the step-count trigger fires early and section 9.1's arithmetic is unforgiving.

What Chapter Nine Establishes

What exists now	What later chapters put through it
Loop with six termination conditions	Graph nodes with per-node budgets *(10.4)*
Typed step records with cost and error	Checkpointed state and resumable runs *(10.5)*
Tool registry with risk tiers	Third-party tools and MCP trust *(12.3)*
Permission chokepoint and approval suspension	The full output policy and human-in-the-loop *(15.1)*
Denied-call signal	Injection detection and alerting *(15.7)*
Compound reliability arithmetic	Deciding graph shape over agent freedom *(10.2)*

Chapter checklist

Can you...	Section
State what an agent is in one sentence, without the word autonomous	9.1
Compute the success probability of a ten-step run at your per-step accuracy	9.1
Explain why halving the steps beats improving the model	9.1
Name six ways a run can stop, and say which one is success	9.2
Say why the budget is checked before the call rather than after	9.2
Give three properties of a schema that constrains rather than invites misuse	9.3
Explain why advertising fewer tools improves quality as well as cost	9.4
Say why a denied call is an observation rather than an abort	9.4
Justify structured state over summarisation	9.5
Explain why one ceiling is insufficient, with a case each way	9.6
Say why permission enforcement belongs in the loop and not in the tool	9.7
Name the situations where reflection helps and where it is worthless	9.8
State the only good reason to add a second agent	9.9
Say what a rising rate of denied calls might mean	9.10

What Chapter 10 builds

Chapter 9's agent decides its own path, which is the source of both its usefulness and its variance. Chapter 10 asks how much of that freedom you actually need, and the answer is usually less than the architecture currently grants.

Explicit graphs with typed state and conditional edges, so the topology is reviewable in a pull request rather than emergent at runtime. Checkpointing after every node, so a run survives a deploy and an approval pause does not hold a worker. Per-node budgets and per-node evaluation, so you can test a stage rather than only a whole run. And the decision procedure for where a model-chosen branch genuinely earns its variance, which is a shorter list than most designs assume.