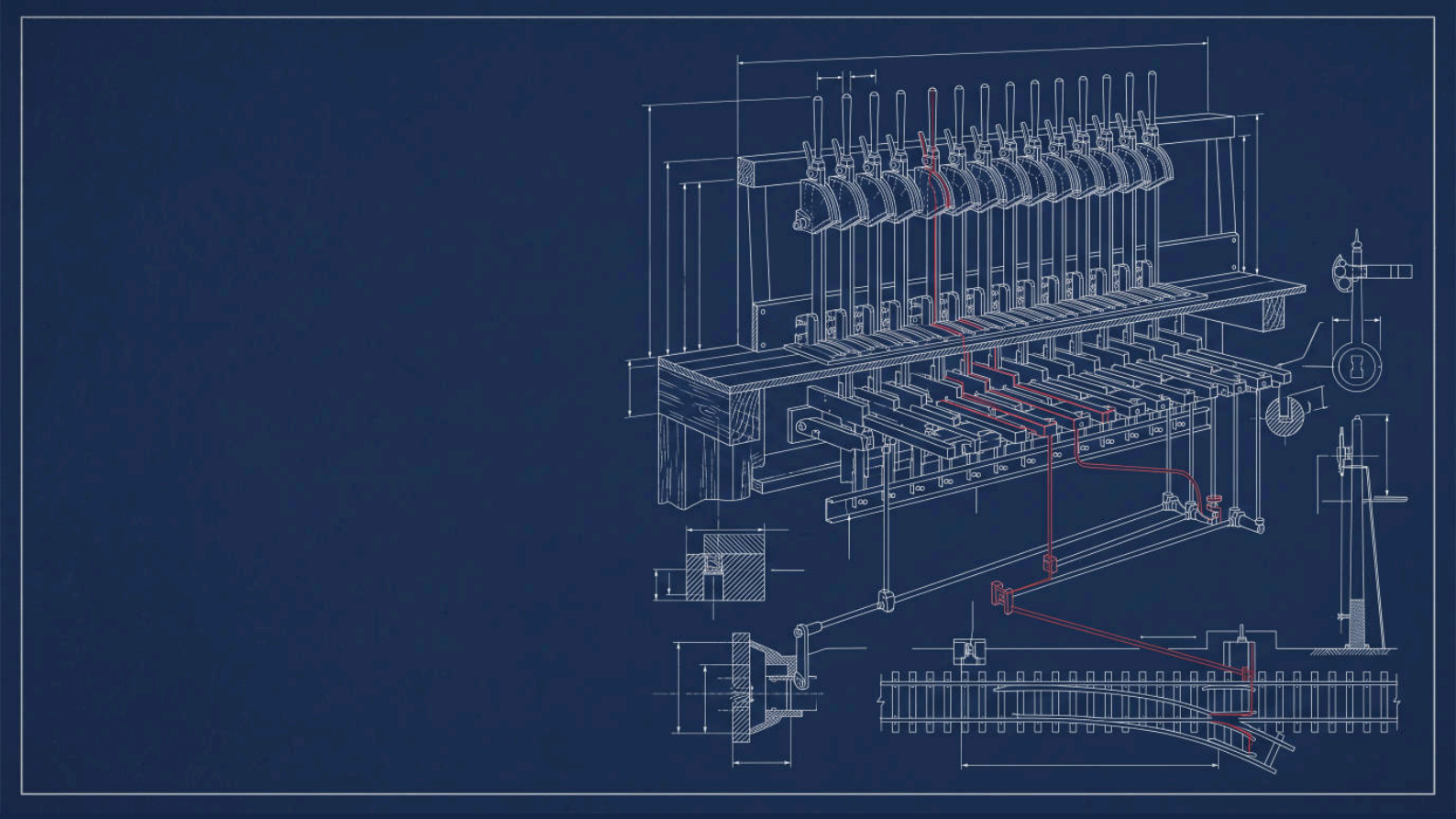

CHAPTER TEN

Orchestration with Explicit Graphs

SECTIONS 10.1 TO 10.11

Chapter 9's agent decides its own path, which is the source of both its usefulness and its variance. This chapter asks how much of that freedom you actually need. Usually less than the design grants.

CHAPTER TEN

Contents

PART III · AGENTS

Chapter 9's agent decides its own path, which is the source of both its usefulness and its variance. This chapter asks how much of that freedom you actually need. The answer, for most work, is one or two branches rather than an open loop, and the difference is measurable in cost, testability and how a run behaves when a deploy lands mid-flight.

10.1 Why Graphs Beat Free-Form Agents

Section 9.1 established the arithmetic: reliability compounds downward, so every step removed from the loop multiplies rather than adds. A graph is how you remove steps without removing capability.

> **The definition worth holding: a graph is a topology you declare, and an agent is a topology the model discovers at runtime.** Everything that follows in this chapter is a consequence of that one difference. A declared topology can be read in a pull request, drawn without executing anything, validated before it runs, checkpointed between nodes, and tested a stage at a time. A discovered one can do none of those things.

Four properties follow, and each is something Chapter 9's loop structurally could not provide.

Property	Why the graph has it and the loop does not
Startup validation	The set of nodes and edges exists before the run, so unreachable nodes and paths that never terminate are caught at boot rather than discovered in a trace.
Checkpointing	Node boundaries are natural commit points. A free-form loop has no boundary that is meaningful to persist between.
Per-node budgets	A global ceiling says a run went wrong. A per-node limit says *which part*, which is a diagnosis rather than an alert.
Per-node evaluation	Section 10.10. You can measure retrieval separately from answering, so a regression has an address.

10.2 Deciding Where Freedom Earns Its Variance

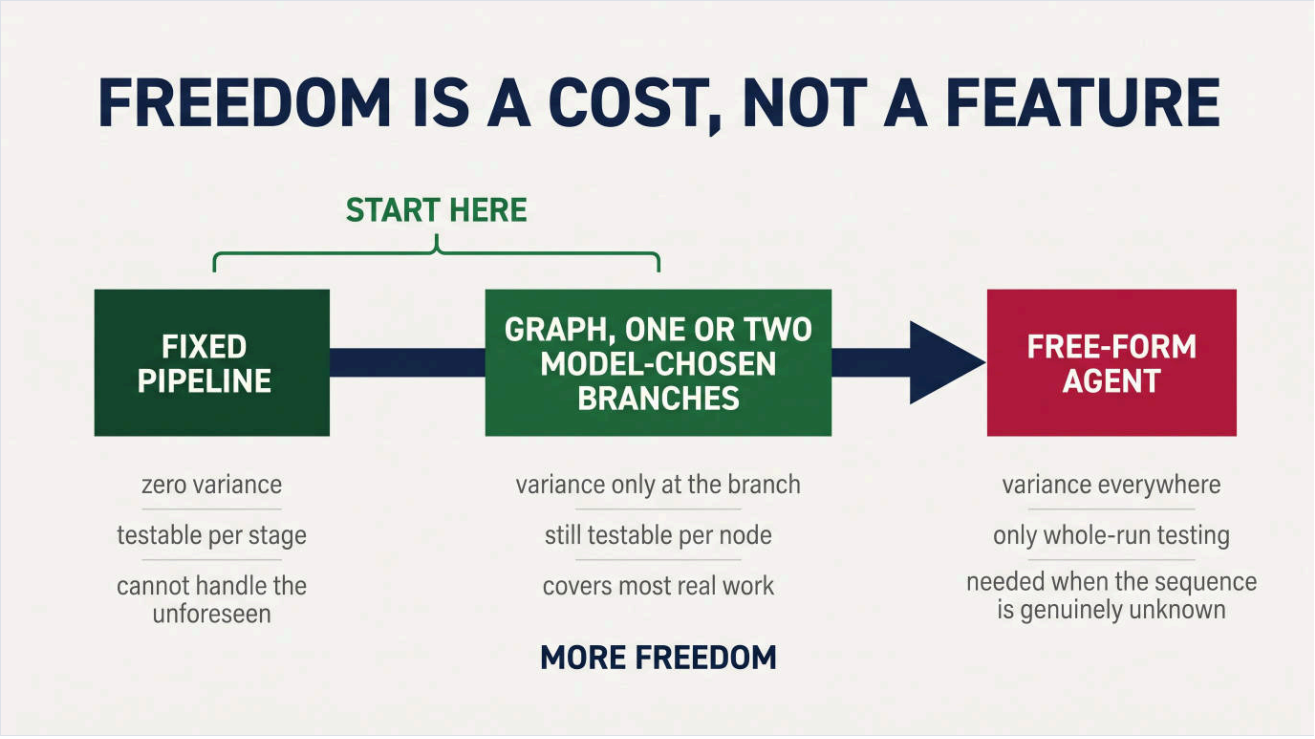

Figure 10.1 Three points on a spectrum. Most production work belongs at the left or the middle, and most designs start at the right.

The instinct is to grant freedom and constrain it later. The better default is to grant none and add it where a measurement says it is needed, because **freedom is a cost paid on every run and a capability used on few of them.**

Three questions decide whether a given decision point should be model-chosen.

Can the choice be made from the state without a model? "Do we have enough evidence" is often a threshold on a retrieval score, and a threshold is free, deterministic and testable. If a comparison would do, use the comparison.

Is the set of destinations finite and known? If yes, the model's job is classification into a small set, not open-ended planning. That is a far easier task, it can be done by a small model per section 3.6, and it can be evaluated as a

classifier.

Would a wrong choice be visible? A branch whose wrong path produces a plausible answer is a branch you cannot debug. Prefer branches where the failure is loud.

> THE SHAPE MOST DOCUMENT PRODUCTS CONVERGE ON
>
> A fixed spine with one or two branches: retrieve, decide whether the evidence suffices, answer, verify, and abstain if verification fails. The only genuinely model-chosen decision is the sufficiency judgement, and even that is often a score threshold with a model consulted only near the boundary.
>
> That is not a lesser architecture than an agent. It is the same capability with the variance removed from every decision that did not need it, which is all of them but one.

10.3 Nodes, Edges and Typed State

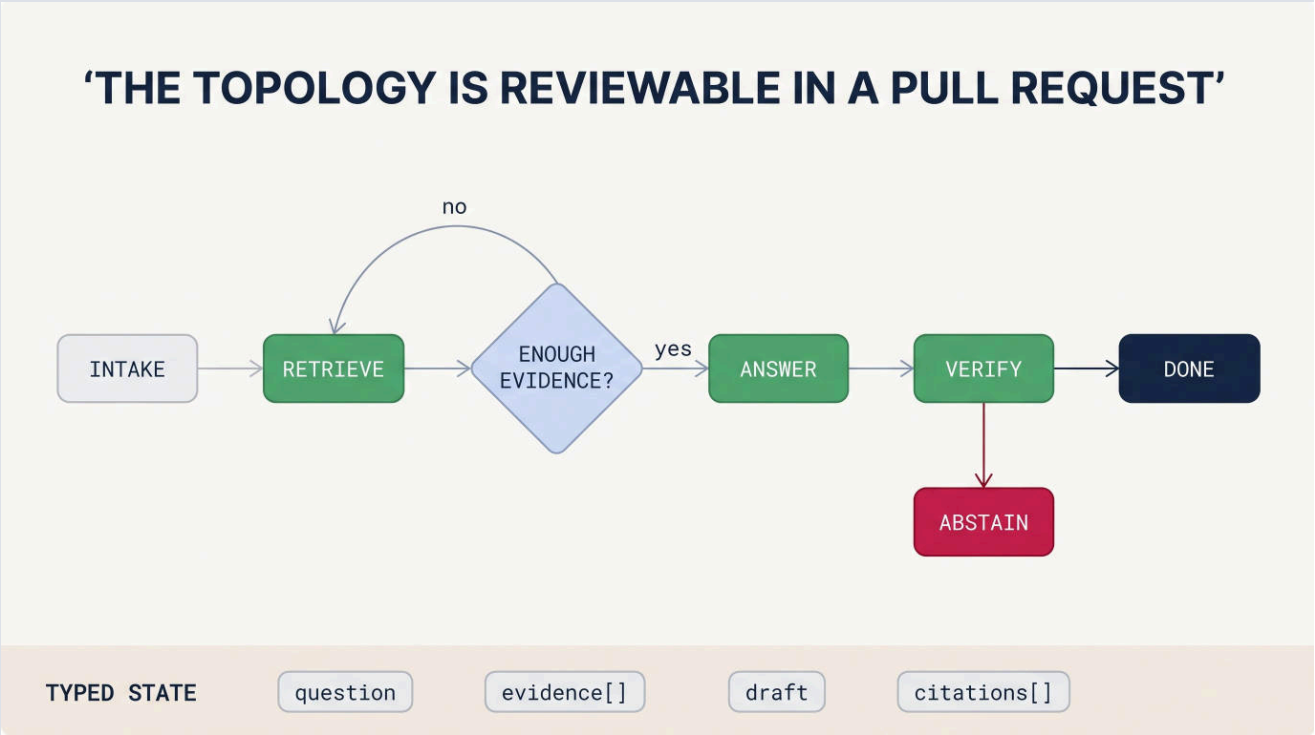

Figure 10.2 Nodes transform typed state; edges decide what runs next; the branch is the one place judgement enters.

Three objects, and the constraints on each are what make the whole thing work.

```
# src/docassist/graph/engine.py
@dataclass(frozen=True)
class Node(Generic[S]):
    name: str
    run: Callable[[S], Awaitable[S]]
    # Per-node limits. A global budget stops the run; a per-node limit tells
    # you WHICH part was stuck, which is the difference between an alert and
    # a diagnosis.
    max_visits: int = 3
    max_cost_usd: float | None = None
```

A node takes state and returns state. Not a message, not a string: a typed object. Section 9.5 argued for structured state over summarisation inside an agent; here it is enforced by the signature. The state is what gets checkpointed, so anything not in it does not survive a restart.

State is replaced, not mutated. Nodes return a new object, which is what makes a checkpoint a value rather than a snapshot of something still moving. In the repository's example the state is a frozen dataclass and nodes use `replace()`.

```
@dataclass(frozen=True)
class QA:
    question: str
    evidence: tuple[str, ...] = ()
    draft: str | None = None
    approved: bool = False
    rounds: int = 0
```

An edge is a function from state to a node name. A static edge ignores the state; a branch consults it. Either way the return value is a name, never free text, so the reachable set is finite.

Validation at startup, not discovery at runtime

```
def validate(self) -> None:
    """Run at startup. Every failure caught here is one that would otherwise
    surface mid-run, after money has been spent."""
    if self.entry not in self._nodes:
        raise ValueError(f"entry node {self.entry!r} is not defined")

    for name in self._nodes:
        if name not in self._edges:
            raise ValueError(f"node {name!r} has no outgoing edge")

    for frm, dests in self._dests.items():
        for d in dests:
            if d != END and d not in self._nodes:
                raise ValueError(f"edge {frm} -> {d!r} targets no node")

    reachable, frontier = {self.entry}, [self.entry]
    while frontier:
        for d in self._dests.get(frontier.pop(), set()):
            if d != END and d not in reachable:
                reachable.add(d); frontier.append(d)
    if orphans := set(self._nodes) - reachable:
        raise ValueError(f"unreachable nodes: {sorted(orphans)}")

    if not self._can_reach_end():
        raise ValueError("no path from entry reaches END")
```

Two of those checks earn their place immediately:

```
startup catch: unreachable nodes: ['orphan']
startup catch: no path from entry reaches END
```

An orphaned node is usually a refactor that removed the last edge into it, and it is invisible until someone asks why a feature stopped working. A graph with no path to termination is a run that will exhaust its budget every time. Both are cheap to detect and expensive to find later.

> THE CHECK THAT IS MISSING ON PURPOSE
>
> There is no check that every cycle terminates, because that is undecidable in general. The graph substitutes something weaker and practical: a per-node visit counter (section 10.9) that bounds every cycle whether or not its condition is well founded. Prefer a bound you can enforce to a proof you cannot.

10.4 Conditional Routing and Per-Node Budgets

The branch is where the model's judgement enters, and the constraint on it is that its output space is declared.

```
def branch(self, frm: str, choose: Callable[[S], str],
           destinations: set[str]) -> "Graph[S]":
    """Destinations are DECLARED, so the set of reachable targets is finite,
    reviewable, and checkable before the graph ever runs."""
    def guarded(s: S, choose=choose, destinations=destinations) -> str:
        nxt = choose(s)
        if nxt not in destinations:
            raise ValueError(
                f"branch from {frm} returned {nxt!r}, "
                f"not one of {sorted(destinations)}")
        return nxt
    self._edges[frm] = guarded
    self._dests[frm] = set(destinations)
    return self
```

The guard is doing two jobs. At runtime it refuses a destination the model invented, which turns a class of hallucination into a caught error rather than an undefined jump. At startup the declared set feeds the reachability check above, so

the topology can be analysed without executing a single node.

```
async def test_a_branch_returning_an_undeclared_target_is_refused():
    """The finite destination set is the whole point of a declared branch."""
    g.branch("a", lambda s: "elsewhere", {END})
    out = await g.run(QA("q"), Budget())
    assert out.stop is Stop.NODE_FAILED
    assert "not one of" in out.steps[-1].error
```

Budgets at two levels

Chapter 9 gave the run three ceilings. The graph keeps those and adds a per-node cost limit, because the two answer different questions.

Limit	Answers
Run: `max_steps`, `max_cost_usd`, `deadline_s`	Should this run continue at all?
Node: `max_visits`, `max_cost_usd`	Is *this stage* behaving as designed?

A node that quietly costs ten times its budget is a bug worth catching even when the run as a whole stays inside its ceiling, because it will not stay inside it when the corpus grows. The per-node ceiling turns a future incident into a present test failure.

10.5 Checkpointing and Resumable Runs

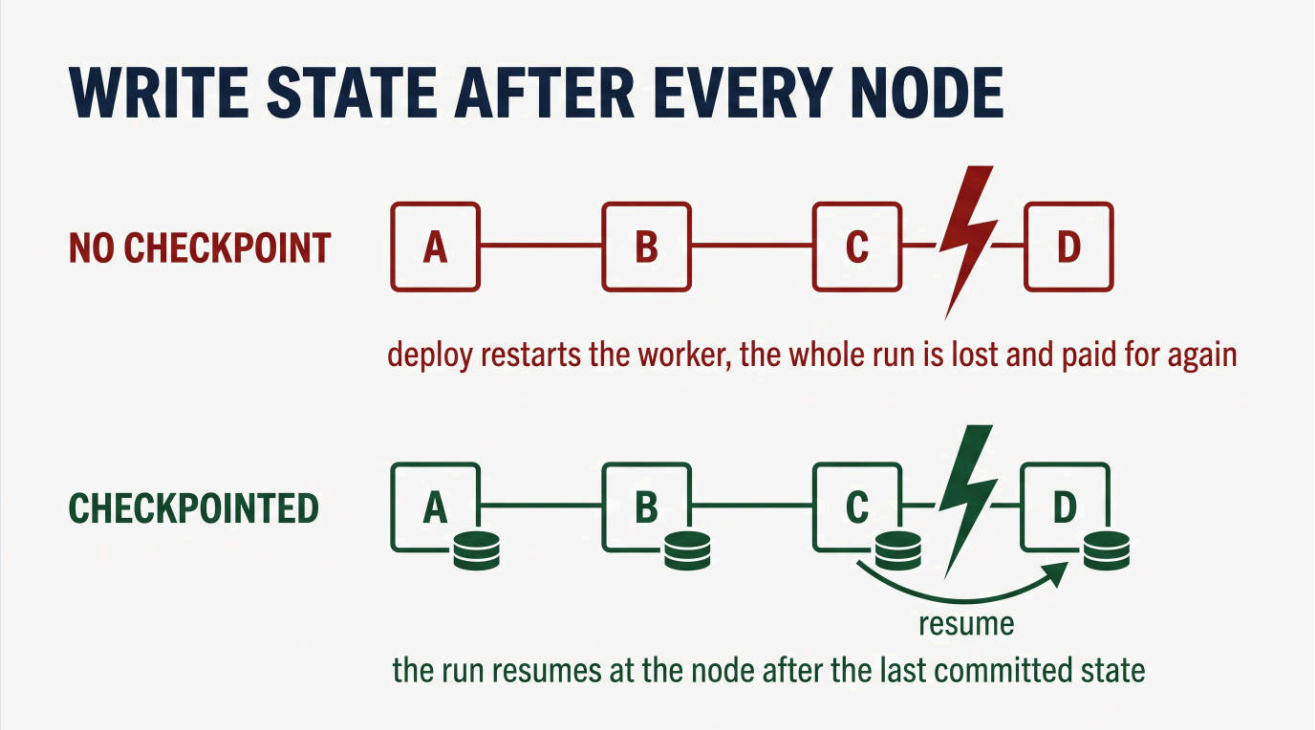

Figure 10.3 A deploy lands mid-run. One of these architectures loses the work and pays for it again.

Chapter 8 established that a worker restart is not hypothetical: it happens on every deploy, every scale-in, and every out-of-memory kill. Chapter 9's agent loses its entire run to any of them. A graph does not have to.

```
# Commit AFTER the node succeeds and BEFORE the next one starts.
# A crash between the two replays one node, which is why nodes must be
# idempotent (section 8.10).
if store is not None:
    await store.put(Checkpoint(run_id, nxt, state, visits,
                               budget.steps_used, budget.cost_used))
node_name = nxt
```

The comment carries the whole design. Committing after the node means a crash costs at most the node in flight. Committing before the next node starts means the checkpoint names *where to resume*, not where it was. And the gap between the two is a real window: a crash there replays one node, so **nodes must be idempotent**, which is section 8.10's argument arriving one level up.

Note what the checkpoint carries beyond the state: `visits`, `steps_used` and `cost_used`. Without those, a resumed run gets a fresh budget and a fresh loop guard, so a run that crashes repeatedly can spend without limit while every individual attempt looks well behaved.

```
async def resume(self, run_id: str, store: CheckpointStore[S],
                 budget: Budget | None = None) -> RunResult[S]:
    cp = await store.get(run_id)
    if cp is None:
        raise KeyError(f"no checkpoint for {run_id}")
    b = budget or Budget()
    b.steps_used, b.cost_used = cp.steps_used, cp.cost_used     # carried over
    return await self.run(cp.state, b, run_id=run_id, store=store,
                          start_at=cp.next_node, visits=cp.visits)
```

Resuming across a deploy

The headline of this section is that a run survives a deploy, and a deploy is exactly the event that may have changed the graph. The checkpoint's `next_node` is a name, and a name only means something against a topology. Rename the node, remove it, or rewire its edges, and the checkpoint now points into a graph that no longer exists, or worse, into one where the name survives but the meaning does not.

The control is to stamp every checkpoint with a topology version. A hash of the declaration is enough: the node names and the declared destination sets, which are the same objects `validate()` already reads at startup.

```
def topology_version(self) -> str:
    surface = sorted((name, tuple(sorted(self._dests[name])))
                     for name in self._nodes)
    return sha256(repr(surface).encode()).hexdigest()[:12]

# In resume(), before anything runs:
if cp.topology_version != self.topology_version():
    return self.on_topology_change(cp)   # a declared policy. Never proceed.
```

On resume, a matching version proceeds. A changed version hits a policy you declared before the deploy, and there are three defensible ones. **Drain:** in-flight runs finish on the old topology, which means keeping the old graph deployed until its checkpoints are gone; this is the default for graphs that change often. **Migrate:** an explicit mapping from old node names to new ones, written by the person who changed the graph and reviewed the way a schema migration is reviewed, because that is what it is. **Abort:** terminate the run, refund or notify, and let the request be resubmitted against the new graph; correct when runs are cheap and migration mappings are not worth their review burden.

What is never acceptable is the silent fourth option, which is what a version-free checkpoint gives you by default: resume onto whatever graph happens to be deployed and hope the names still line up. The lucky failure is a `KeyError` on a removed node. The unlucky one is a name that still exists with different edges, a run that proceeds down a path nobody designed, and an incident report that starts from a trace no current source file explains.

10.6 Agent Runs as Durable Jobs

Chapter 8 built a queue whose jobs are short and idempotent. A graph run is neither by default, and reconciling the two is a matter of choosing where the job boundary sits.

Job boundary	Consequence
One job per run	Simple. The visibility timeout must exceed the longest run, and section 8.10 showed why a long timeout is bad: a genuinely dead worker holds its job hostage for that whole window.
One job per node	Each job is short, so the visibility timeout can be tight. The run becomes a chain: each node's job enqueues the next. Costs an enqueue per node and makes the trace a chain rather than a tree.
One job per segment	The pragmatic middle. Break at interrupts and at long-running nodes; run the rest inline. This is what most systems arrive at.

The segment version is small enough to show whole. A segment job resumes from the checkpoint, runs until the graph stops, and enqueues the continuation if there is more to do:

```
# The segment budget's deadline is the boundary: shorter than the queue's
# visibility timeout, well inside the run's own ceilings.
async def handle_segment(job: Job) -> None:
    out = await graph.resume(job.run_id, store, segment_budget(job))
    if out.stop is Stop.INTERRUPTED:
        return          # a human resumes this one, not the queue (10.7)
    if out.stop is Stop.DEADLINE and run_budget_remains(job):
        await queue.enqueue(Priority.INTERACTIVE,    # the run's class of service (8.9)
                            Job(run_id=job.run_id))  # the chain continues
```

The checkpoint store from section 10.5 is what makes the chain safe: the enqueued job carries only the `run_id`, because everything else lives in the checkpoint, and a redelivered segment job re-reads the checkpoint and repeats at most the node in flight. The segment boundary is therefore always a checkpoint boundary, never a point inside a node.

Whichever you choose, the **idempotency key from section 8.10 becomes a key per node, not per run**: `tenant | run_id | node | attempt-invariant-hash`. Keying per run means a redelivery after node three replays nodes one to three, which is exactly the work the checkpoint exists to protect.

10.7 Human-in-the-Loop Interrupts

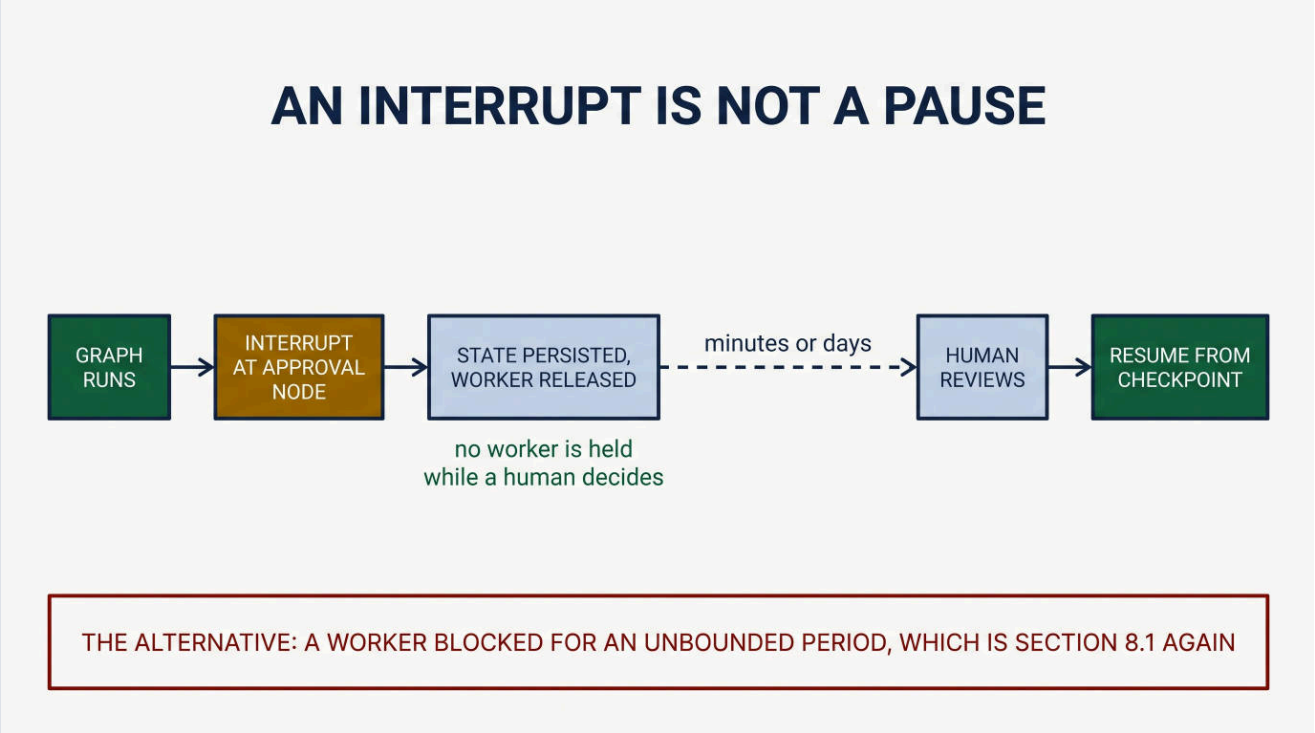

Figure 10.4 The distinction that keeps a worker pool alive when approvals take hours.

Section 9.7 gated irreversible actions behind approval and suspended the run. The graph makes that a first-class control flow event.

This is **human-in-the-loop** as an architectural pattern: a point where the automated flow stops, deliberately and by design, for a decision the system is not entitled to make alone. This section builds the machinery that makes the stop affordable. Section 15.5 designs the other half, the approval surface the person actually sees, and what makes their review a control rather than a formality.

```
class Interrupt(Exception):
    """Raised by a node that needs a human before it can proceed.

    Not an error. The run persists its state, releases the worker, and is
    resumed later by resume(). Blocking instead would hold a worker for an
    unbounded period, which is section 8.1 again.
    """
```

Three properties make this workable, and the repository demonstrates all three.

```
interrupt       -> interrupted  payload={'draft': '60 days'}  resume_at=approve
after approval  -> completed    rounds=2 (retrieval not redone)  path=['approve', 'verify']
```

The worker is released. The run is a row in a store, not a held thread. An approval that takes three days costs nothing while it waits.

The checkpoint resumes *at* the interrupted node, not after it, so the node runs again with the human's decision now present in the state. That is why the approval node re-executes and finds `approved=True` rather than being skipped.

Work already done is not redone. The resumed run shows `rounds=2`, meaning both retrieval passes survived, and its path is only `['approve', 'verify']`. A design that restarts on approval pays for the whole run twice and, worse, may retrieve different evidence the second time, so the human approved something other than what ships.

Two exits belong in the same machinery. A cancellation, whether from an operator killing a misbehaving run or a user withdrawing the request, is a stop reason like any other: it releases the worker, marks the checkpoint terminal so nothing resumes it, and records who cancelled and why in the trace. Treating cancel as an exception to be swallowed leaves a resumable checkpoint behind, and someone will resume it.

An interrupted run is also not entitled to wait forever. An approval that never comes is the common case, not the edge case, so every interrupt carries a TTL and an expiry policy: after seven days, abort and notify, or escalate to a different approver. Without one, the checkpoint store accumulates zombie runs that hold no worker but still hold state, budget headroom and, eventually, an auditor's attention. Expiry is a scheduled sweep over checkpoints, which Chapter 8's machinery already knows how to run.

> THE INTERRUPT PAYLOAD IS THE APPROVAL UI
>
> Section 9.7 warned that "the agent wants to call send_email, approve?" trains people to click yes. The payload is where you fix that: carry the rendered effect, the evidence used, and the cost so far. An interrupt that carries a tool name and nothing else has moved the theatre into a better architecture.

10.8 Parallel Branches and Fan-In

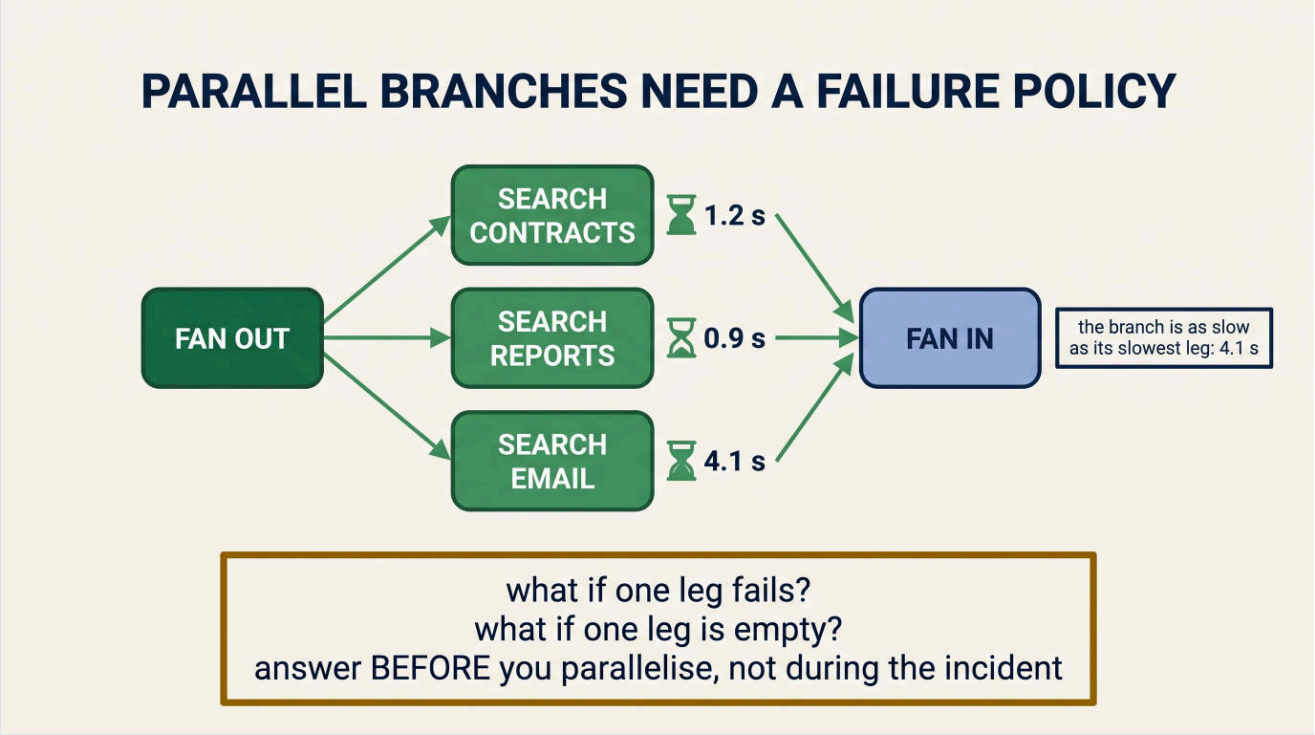

Figure 10.5 Three legs, one of them slow. The interesting questions are about the failures, not the speed-up.

Independent nodes can run concurrently, and the latency win is real: three searches at 1.2, 0.9 and 4.1 seconds cost 4.1 seconds in parallel rather than 6.2 in sequence. The win is smaller than it looks, because **a parallel branch is as slow as its slowest leg**, and it is the slowest leg that varies.

The questions that matter are not about speed:

Question	Decide before you parallelise
One leg fails	Fail the branch, or continue with partial results and record which leg is missing? For retrieval, usually continue. For a compliance check, usually fail.
One leg is empty	Is empty a result or an error? Section 9.4's tool-description problem, at branch level.
One leg is slow	A per-leg deadline shorter than the branch deadline, so one slow source cannot consume the whole budget.
Legs disagree	Fan-in needs a merge rule. For retrieval this is reciprocal rank fusion from section 7.6, which is exactly this problem solved once already.

The rule I would apply: **parallelise reads freely, and never parallelise writes.** Concurrent read legs are independent and their failure modes compose cleanly. Concurrent writes reintroduce ordering questions that the graph existed to remove, and the latency saved is rarely worth the class of bug bought.

The fixed legs generalise to a map over a collection: one leg per retrieved document, one per invoice line, with *N* known only at runtime. This looks like the topology escaping its declaration, and it need not be. What is declared is one node template applied per element, a bound on *N*, and one merge rule at the fan-in. The shape is still reviewable in a pull request; only the multiplicity is decided by the data.

The bound and the failure policy are the parts that must be declared up front, because they are the parts a runtime value will otherwise decide for you. An unbounded map over "all matching documents" is a cost ceiling breach waiting for a large tenant, so cap *N* and state what happens above the cap: process the top *N* by relevance and record the truncation, per section 2.10. And the single-leg failure question from the table above now needs a quantified answer, because at *N* of forty some leg will fail on most runs: all legs must succeed, a quorum suffices, or best effort down to a declared floor below which the merge refuses to run. Choose per fan-out, and test the floor the way section 10.10 tests any other node.

10.9 Cycles, Retries and Loop Guards

Figure 10.6 A global budget stops the run. A per-node counter says which loop was stuck.

Cycles are the reason to use a graph rather than a pipeline. "Retrieve, assess, rewrite the query, retrieve again" is a genuine loop and section 7.9 showed it is worth having. What it needs is a bound that does not depend on the loop's own condition being correct.

```
visits[node_name] = visits.get(node_name, 0) + 1
if visits[node_name] > node.max_visits:
    return RunResult(state, Stop.LOOP_GUARD, tuple(steps), budget.cost_used)
```

```
async def test_the_loop_guard_names_the_stuck_node():
    """A global budget stops the run. A per-node counter says which loop."""
    out = await g.run(QA("q"), Budget(max_steps=100))
    assert out.stop is Stop.LOOP_GUARD
    assert len(out.steps) == 3        # stopped by the node, not the budget
```

That assertion is the point. With `max_steps=100` the run had ninety-seven steps of headroom, and it stopped at three because the node said so. The stop reason names the node, so the trace answers "which loop" without anyone reading the steps.

Set `max_visits` from the successful runs, not the possible ones. If ninety-five percent of successful runs visit the retrieval node once and the rest twice, a limit of three is generous. A limit of ten is a budget leak with no quality upside, and it delays the signal you actually want, which is that this loop is not converging.

Two failure modes worth distinguishing. A loop that *oscillates* between two states is usually a branch condition reading a field the loop never changes. A loop that *progresses but never satisfies* its condition is usually a threshold set where

the evidence cannot reach, and the fix is the abstain path from section 6.9 rather than more iterations.

10.10 Per-Node Evaluation

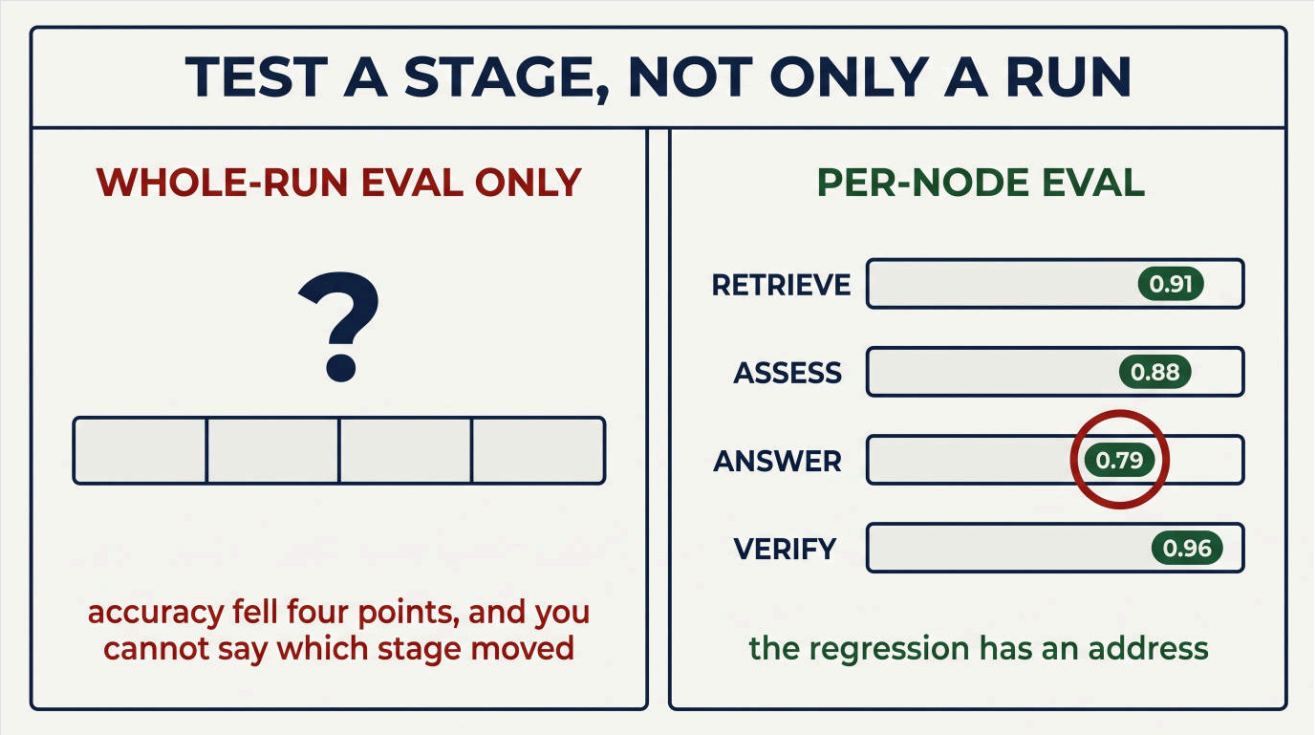

Figure 10.7 Whole-run accuracy tells you something moved. Per-node scores tell you what.

This is the capability that most justifies the architecture, and it is the direct descendant of section 7.2's failure taxonomy. There the diagnosis was manual and after the fact; here it is continuous, because each node has an input type, an output type and therefore a test.

```
# Each node is an independently testable function of typed state.
async def test_retrieve_node_finds_the_clause():
    out = await retrieve(QA(question="what is the notice period?"))
    assert "E1" in out.evidence

async def test_answer_node_abstains_without_evidence():
    out = await answer(QA(question="q", evidence=()))
    assert out.draft is None
```

Three things become possible that whole-run evaluation cannot provide.

A regression has an address. Overall accuracy falling four points is a mystery. The answer node falling from 0.88 to 0.79 while retrieval holds at 0.91 is a work item.

Nodes can be evaluated at different cadences. Retrieval evaluation from section 7.10 is cheap and can run on every commit. An expensive judged evaluation of the answer node can run nightly. Whole-run evaluation forces the cost of the most expensive stage onto every check.

A node can be replaced with confidence. Swapping the retrieval node for the hybrid retriever of section 7.5 is a contained change with a contained test, provided the state contract is unchanged. That is the evolvability axis from section 1.1, finally paid out.

> KEEP WHOLE-RUN EVALUATION TOO
>
> Per-node scores can all improve while the run gets worse, for the same reason section 7.10 gave about recall and answers: the stages interact. A retrieval node tuned for recall feeds an answer node that then dilutes. **Per-node evaluation localises a regression; whole-run evaluation detects one.** You need both, and the run-level metric is the one that gates a release.

10.11 Three Orchestration Architectures Compared

GOOD

The fixed pipeline

```
evidence = await retrieve(question)
draft    = await answer(question, evidence)
return await verify(draft, evidence)
```

Three calls in sequence. Zero variance, trivially testable, and correct for a very large fraction of document work. Section 9.1's arithmetic favours it: three deterministic stages at 95 percent give 0.857, where a five-step agent at the same per-step accuracy gives 0.774.

What it leaves unsolved: nothing adapts. A question needing a second retrieval pass gets one pass. A question the evidence cannot answer gets an answer anyway, unless you bolt on the abstain path. And there is no place to put a human.

BETTER

The declared graph

The engine of this chapter: typed state, declared edges, one or two branches, validated at startup, checkpointed between nodes, interruptible for approval, bounded per node.

What it still does not solve. The topology is fixed at deploy time, so a genuinely novel task shape needs a code change. Parallel branches need a hand-written merge rule per fan-in. And the graph itself is now an artifact that can drift from what the evals cover, which is the section 5.8 problem arriving for topology.

BEST

Graph with a bounded agent inside one node

The arrangement I would defend for most systems that genuinely need open-ended behaviour: a declared graph for the spine, with *one* node containing Chapter 9's bounded agent loop, restricted to read-only tools and a tight step budget.

The graph provides checkpointing, approval, per-node evaluation and the write permissions. The agent node provides open-ended exploration where the sequence truly cannot be known. Crucially the agent's blast radius is one node: it cannot skip verification, cannot reach an irreversible tool, and cannot consume the run's whole budget.

Axis	Good: pipeline	Better: declared graph	Best: graph with agent node
Adaptivity	None	Declared branches	Open, inside one node
Variance	Zero	Bounded to branches	Bounded to one node
Survives a deploy	No	Resumes from checkpoint	Resumes from checkpoint
Human approval	Nowhere to put it	Interrupt node	Interrupt node
Testability	Per stage	Per node	Per node, agent node whole-run only
Build cost	Days	Two to three weeks	Three to five weeks

Durable-workflow engines and graph frameworks solve checkpoint and resume wholesale, and adopting one is often the right call: the persistence, the replay semantics and the operational tooling are exactly the undifferentiated plumbing you should not enjoy rebuilding. The reason to understand the hand-built version is that the plumbing was never the content of this chapter. The failure policies were: what a checkpoint must carry, what happens when the topology changes under a resume, where the segment boundary sits, who bounds a fan-out and what an interrupt owes its TTL. Every one of those remains yours to declare in any engine, and an engine adopted without declaring them is the silent fourth option from section 10.5 with a vendor logo on it.

PROMOTION TRIGGERS

Good to better, when any of these hold: a run must survive a deploy; an approval step exists or is coming; the same question sometimes needs a second pass; or you cannot tell which stage caused a quality change. The first two are hard requirements rather than preferences, and they arrive earlier than teams expect.

Better to best, only when you can point at real traffic the declared branches cannot serve. "*The sequence cannot be known in advance*" *is a claim to verify, not assume.* Sample fifty runs and ask how many took a path your graph does not already contain. In most document products the answer is a handful, and they are better served by an abstain path than by an agent.

The long-horizon run: planner, builder, evaluator

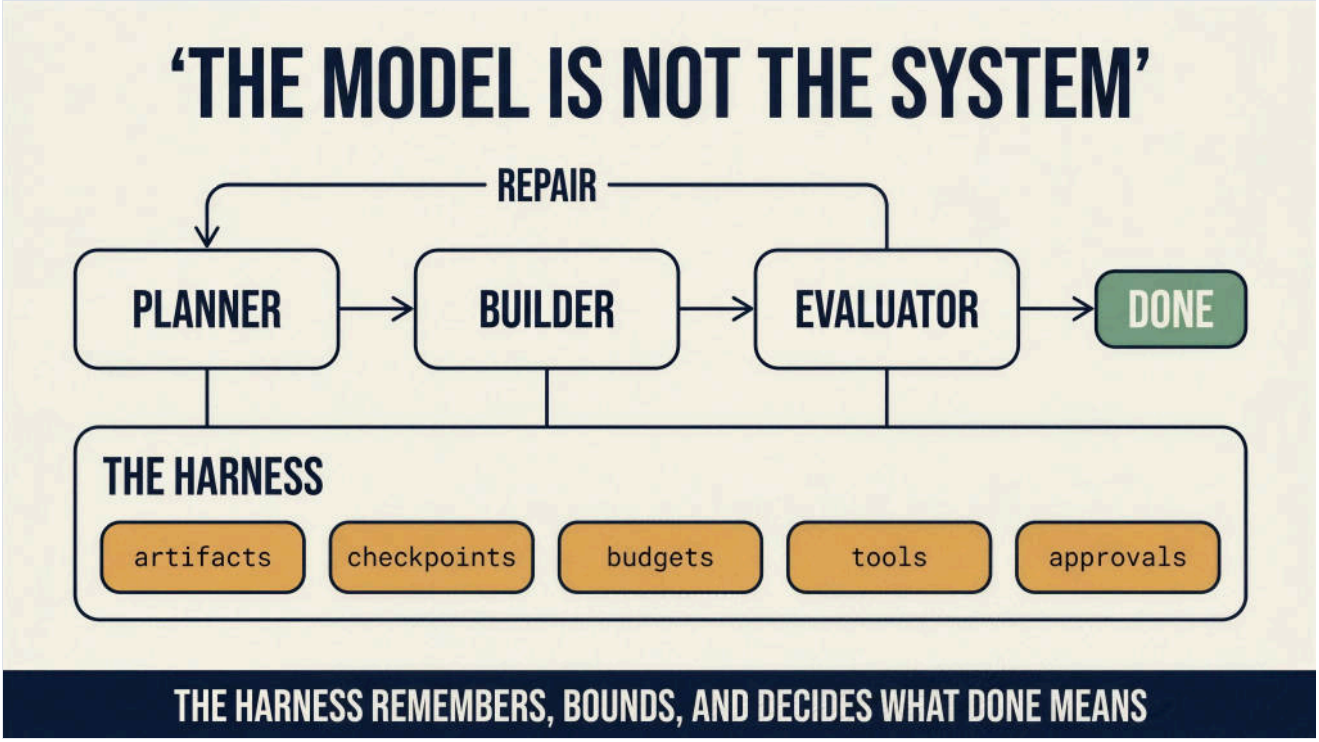

Figure 10.8 A small software team running inside a sandbox, and the harness that makes it one system.

There is a class of work this chapter's machinery was quietly built for, and by 2026 it has a name in every framework's marketing: the **long-running agent**, a system that works autonomously for hours, building an application, migrating a codebase, running QA across hundreds of files. The published architectures for this work, including the harness designs the frontier labs describe for their own multi-hour coding sessions, converge on the same shape, and it is worth seeing that the shape is this chapter run at longer range: a small software team inside a sandbox, with three roles and a harness.

The three roles. A *planner* turns a short goal into a task graph: a product plan, a task list, an ordering. That artefact is a declared topology produced by a model, which is exactly section 10.3's situation, and it earns the same treatment: a finite, reviewable structure, not a vibe the builder holds in its head. A *builder* executes one task at a time inside an isolation boundary, section 12.2's sandbox, holding only the tools that task declares per section 12.4. An *evaluator* checks the result against concrete criteria before the work counts. The split is legitimate by section 9.9's own rule, because each seam is a nameable boundary: the planner and builder need different tools, the builder needs containment, and the evaluator must be outside the builder's context or it is section 9.8's self-approval problem wearing a lanyard. Deterministic checks first, unit tests, linters, policy gates, with a judge only where section 14.3's scorers cannot reach, and calibrated per section 14.4 when one is used.

Artefacts are the memory. A real project holds more context than any window, so the survival mechanism is not a longer prompt and not summarisation, which section 11.3 showed forgets silently. It is forced externalisation: every task ends by writing structured artefacts, what changed, what failed, what remains, into the workspace, the spec, the task list, the implementation notes, the test results. The next task, or the next session after a restart, rehydrates from those artefacts rather than from a transcript. This is section 9.5's structured-state argument promoted to project scale, and it is also why the failure modes of unharnessed long runs are exactly the ones you would predict from this book: doing too much at once (no task granularity), losing the thread (no artefacts), leaving work half-built and undocumented (no forced writes), and declaring victory early (the builder grading its own homework).

The harness is the system. The loop runs plan, build, test, repair, repeat, and everything that makes it survivable is the machinery already on the table: per-task budgets from section 10.4, a checkpoint after every task from section 10.5 with the topology version stamped, resume that never guesses, section 10.7's interrupt wherever a step needs a human signature, and section 10.9's guard on the repair cycle, because build-test-repair is a cycle and an unguarded cycle is a bill. The model contributes none of this. The harness decides what the model sees, which tools it holds, where progress is written, and, through the evaluator, what "done" means. **"Done" is the evaluator's word, not the builder's**, which is the grounding rule of section 6.9 applied to work instead of answers. Frameworks now ship the plumbing for all of it, checkpointing, pause and resume, approval, handoffs; the build-versus-adopt judgement two paragraphs up applies unchanged, because the declarations, the budgets, the criteria and the boundaries remain yours in any of them.

What Chapter Ten Establishes

What exists now	What later chapters put through it
Typed state checkpointed per node	Conversation and durable memory as state fields *(11.4)*
Interrupt and resume	Approval queues and human handoff *(15.1)*
Declared destination sets	Tool permissions scoped per node *(12.4)*
Per-node budgets and loop guards	Cost attribution per stage *(16.2)*
Per-node evaluation	The eval gate, localised to a stage *(14.6)*
Startup topology validation	Graph changes reviewed like schema changes *(14.5)*

Chapter checklist

Can you...	Section
State the one difference between a graph and an agent, and derive four consequences	10.1
Give the three questions that decide whether a branch should be model-chosen	10.2
Say why nodes return new state rather than mutating it	10.3
Name two topology errors caught at startup, and why each is expensive later	10.3
Explain why a branch declares its destination set	10.4
Say why the checkpoint carries visits and spend as well as state	10.5
Explain why nodes must be idempotent even with checkpointing	10.5
Choose a job boundary for a graph run, and say what it costs	10.6
Say why an interrupt resumes at the node rather than after it	10.7
Give four questions to answer before parallelising a branch	10.8
Explain why a per-node visit counter beats a global step budget	10.9
Say why per-node evaluation does not replace whole-run evaluation	10.10

What Chapter 11 builds

The graph carries state within a run. Chapter 11 is about state that outlives one: what an assistant should remember about a user across sessions, and what it must be able to forget.

Four kinds of memory with different lifetimes and different failure modes, and the retrieval problem that memory actually is. Why a summarised conversation is a lossy compression with no error signal, and what to use instead. The write path, which is where most memory systems go wrong: deciding what is worth remembering is a judgement call made on every turn, and a system that remembers everything is a system whose context is full of noise by week three. And the deletion requirement from section 6.10, now applied to inferences a model made about a person rather than documents they uploaded.