Clausey

Durable Agents on Plain Postgres

How we built long-running autonomous workflows for FP&A, operations, and compliance-heavy work using disposable compute, durable database state, and a cron safety net.

LG Larry Galan C Claude (Anthropic) July 2026

THE ARGUMENT IN FOUR LINES

The work	A new class of AI work runs for hours across thousands of items — categorize 5,000 invoices, migrate 200,000 records. It's a coworker finishing a job, not a chatbot answering a turn.
The idea	Keep every piece of state in Postgres and make the compute disposable. A worker can die at any line and lose nothing, because its progress was never inside the worker.
The payoff	Mounted on that engine, an AI agent inherits crash recovery, retries, budgets, and an audit trail by construction — no separate agent infrastructure to build, operate, or trust.
The ceiling	It scales across essentially all agentic back-office work on infrastructure you already run — and where the limits are, they're known, named, and far away.

PART I

The work

Before any architecture, be concrete about what we're trying to make possible — and why today's stacks quietly fail at it.

This is a coworker, not a chatbot

The most valuable thing an AI system can do inside a company usually isn't answer a question. It's *finish a job* — a large, repetitive, consequential job that used to occupy a team for a week. Consider the actual work:

01 Go through all **5,000 invoices** — categorize each one, push it to the ERP, reconcile it against the ledger.

02 Migrate **200,000 records** into the new CRM, transforming and validating each one.

03 Review **10,000 contracts**, extract the terms, flag the risky clauses.

04 Work a backlog of **3,000 support tickets** — triage, draft, update, escalate.

05 Reconcile **a quarter of transactions** and build the audit trail behind every entry.

None of these is a single prompt. Each is thousands of small decisions made in sequence, over minutes or hours, with reads, writes, and calls to outside systems at every step. This is **autonomous work**: you hand over the whole job, not one turn of it — and then you leave.

Why this breaks the moment you try it

Most agent frameworks keep the plan in memory — a Python object graph, a state machine in a library, a queue in Redis. It works right up until the process dies mid-run, and then the plan, the partial results, and the "where was I" pointer all evaporate at once.

For a one-shot chat turn that's survivable — you just retry the turn. But a job that reviews ten thousand contracts runs for *hours*, and the longer it runs, the more certain it becomes that *something* restarts underneath it: a deploy, an autoscaler, an out-of-memory kill, a reclaimed spot instance. When a worker is recycled forty minutes in, an in-memory design loses forty minutes of irreplaceable progress — and often can't even tell you how far it got.

TRUTH KEPT IN THE PROCESS — crash at minute 40

Forty minutes of work existed only in memory, so it's simply gone — the job starts over, and nobody can say which items were already pushed.

TRUTH KEPT IN THE DATABASE — same crash, same minute

Every finished item was already a committed row. A new worker reads the state, sees exactly where minute 40 stopped, and carries on — nothing to reconstruct, nothing lost.

Fig. 1 · The same job, the same crash. The only difference is where the truth lived.

> **Your durability ceiling is wherever you decided to keep the truth.**
>
> IF THE TRUTH LIVES IN A PROCESS, YOUR CEILING IS THAT PROCESS'S UPTIME

PART II

The idea

One move fixes it — and the least exciting database in your stack turns out to be the right place to make it.

So move the truth out of the process. Every claim, every lease, every status transition, every event becomes a durable row in a database. The execution code holds nothing important: it reads the current state, advances it by one small increment, and writes it back.

The database is the only source of truth for your state. The compute is disposable.

THE THESIS

Why Postgres — the boring choice is the radical one

You could keep that truth in a purpose-built system — Temporal, a dedicated workflow engine, a managed queue. Our claim is smaller and more useful: for this entire class of work, a database you already run does the job.

The fair question is "why not the existing tools," so here is the honest version. **Temporal** is excellent at durable execution — but it adds an infrastructure tier, and its replay model wants deterministic workflow code. An LLM loop is the opposite of deterministic, so the model calls end up quarantined in activities and the interesting state ends up in a database anyway; we chose to start there. **River, Graphile Worker, and pg-boss** share our instinct — Postgres as the queue — and any of them could carry the dispatch layer. But they hand you jobs, not the agent-

shaped state this paper is actually about: parked parents, attempt history, spend ledgers, audit trails. And if you run Temporal happily today, keep it — this paper is for teams who would rather not introduce it.

THE BORING CASE

Postgres is already in your stack. There's no new service to deploy, secure, pay for, get paged about at 3 a.m., or explain to an auditor. It's been hardened in production for three decades. Choosing it is the lowest-risk decision in the whole design.

THE TECHNICAL CASE

One transaction claims a job *and* updates its state atomically — no gap where work is lost or double-run. One system of record means no queue and database drifting out of sync. And `SELECT … FOR UPDATE SKIP LOCKED` turns an ordinary table into a correct concurrent queue — and it ships in the box.

WHAT YOU DON'T ADD ~~Redis~~ · ~~Temporal~~ · ~~Kafka~~ · ~~a separate queue~~ · ~~an agent framework~~

What you take on instead: you are operating a small orchestrator yourself. The tuning, the retries, and every failure mode in this paper become your job, not a vendor's — the honest trade is a bill you don't pay for engineering attention you do.

A NOTE ON NOVELTY

There is none here, and that's on purpose. People have been building workflow engines on Postgres for a long time — `SKIP LOCKED` queues are folk knowledge, and every pattern in Part III has shipped somewhere before. We claim assembly, not invention. What we changed is the *shape*: state for steps that think (a parent parked mid-plan, an append-only record of every attempt), governance for steps that spend (budgets nested three deep), and recovery that assumes a step may have already acted on the world. The modification is agent-shaped, not engine-shaped.

One spoiler worth carrying through the plumbing ahead: by Part IV, an AI agent's entire reasoning loop — every thought it has, every tool it calls — will be just another row in the tables the next part builds. That is where all of this is going.

PART III

The engine

How the substrate actually works, and the races it has to win. Nothing in this part knows what an agent is; that is the point. (And if you've built `SKIP LOCKED` queues before, much of this will be familiar — skim to §05, where it stops being.)

SEVEN WORDS, IN PLAIN ENGLISH

`orchestrator`	The planner. It decides which step comes next and records what happened. It never does the heavy work itself.
`worker`	The hands. It picks up queued jobs, does them, and reports back. Any worker can pick up any job.
`claim`	How a worker takes a job: one atomic “this is mine now” write. Two workers can never grab the same row.
`lease`	Ownership with an expiry — a library checkout. A live worker keeps renewing it; a dead one stops, and the job goes back on the shelf for someone else.
`fan-out`	One step exploding into many parallel children — one per invoice, one per contract — so a thousand items don't run single-file.
`dead-letter`	The quarantine shelf. After too many failed tries, a job is set aside for a human to inspect instead of retrying forever.
`cron`	A timer that runs a small check on a schedule — the safety net that eventually notices anything a lost signal left behind.

§01 The shape of the whole system

The entire runtime is two stateless functions. The entire state is a handful of tables. A few scheduled timers — cron jobs — are the safety net. The **orchestrator** resolves a run's dependency graph — which steps must wait for which — into levels, runs each step, and finalizes; the **worker** consumes the parallel work a step fans out, claims it under a lease, and pokes the orchestrator to advance once a step's children are done. The functions are stateless;

the distributed system they form is not — partial failure, duplicate delivery, clock skew, and races are all real, and the invariants that tame them (the rest of this part) live in the gaps between these two boxes.

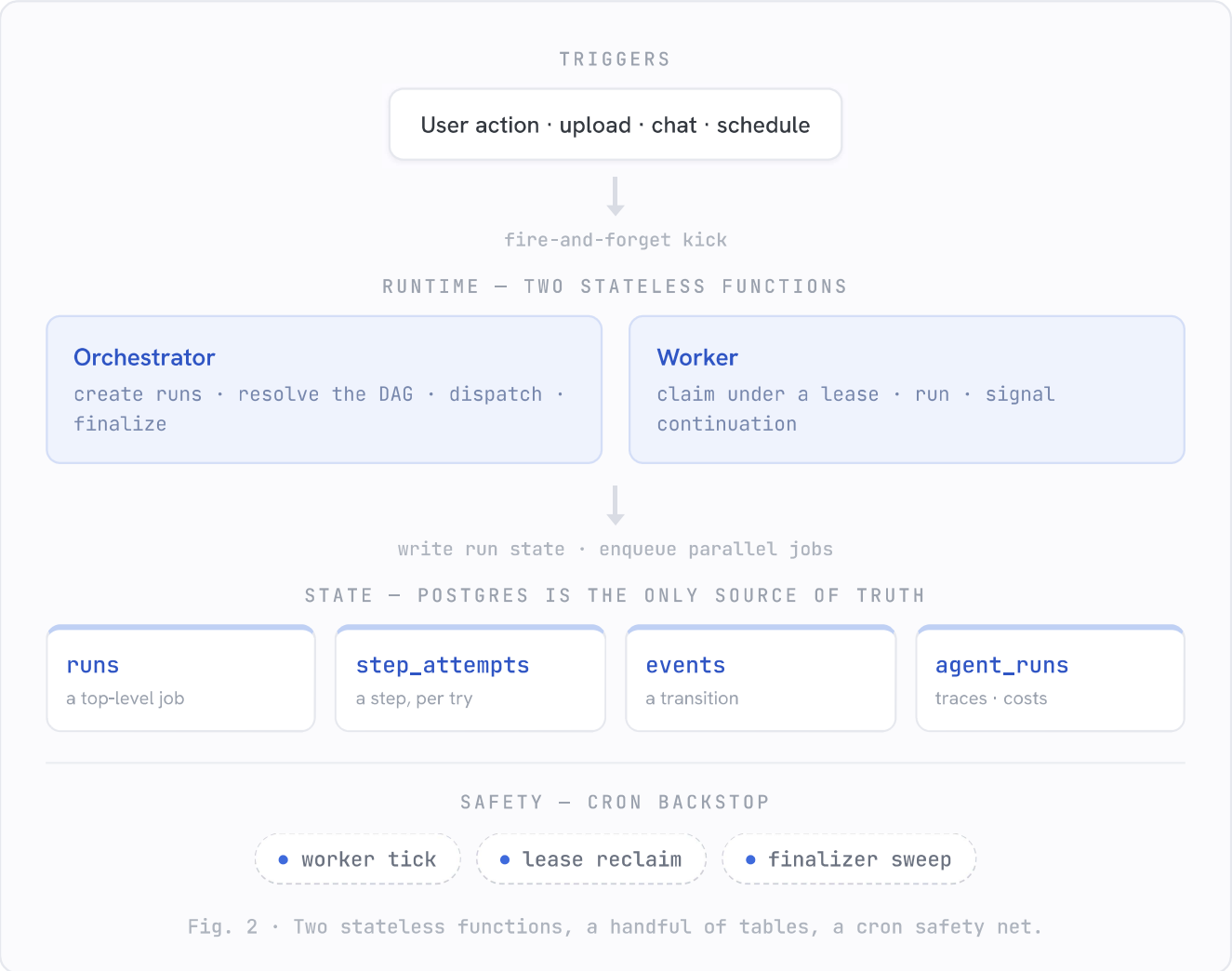


Fig. 2 · Two stateless functions, a handful of tables, a cron safety net.

§02 Pick your grains before your tables

Each table answers a different question at a different grain of resolution. Fold two grains into one and you either bloat a generic table with type-specific columns, or you lose the ability to ask one of the questions.

TABLE	GRAIN	THE QUESTION IT ANSWERS
`runs`	a top-level job	“What executions ran?” One row per run.
`step_attempts`	a step, per try	“What ran inside a run, and can I retry it?”
`events`	a transition	“What changed, in order?” Append-only log.
`agent_runs`	one agent	“What did each agent do?” Master + sub-agents.
`traces`	one block: thinking / text / tool	“What did it think and do?” The flight recorder.
`costs`	one billable model call	“What did each call cost?” The ledger.

§03 The step state machine is the heartbeat

Every unit of work moves through a small, explicit set of statuses — claiming, leasing, finishing, retrying, and reclaiming all turn on these transitions. Transitions move the live attempt row through those statuses — that's the claim you just saw. A *retry*, though, files a fresh attempt row rather than overwriting the failed one, and every transition is also recorded, append-only, in the events log. So any read of “current state” collapses attempts with a status-priority tiebreaker, not a naive `MAX(attempt)`.

`queued` → `claimed` → `running` → `succeeded` `failed / dead-letter`

`awaiting_children` `awaiting_approval`

AND THE THREE WAYS BACK

`claimed → queued`
The worker died before reporting anything. Its lease lapses, and the sweep simply un-claims the row — no one has to notice the death for the job to survive it.

`failed → queued, attempt +1`
Retried after a growing pause — and at the ceiling, `dead_letter`: set aside for a human, never an infinite loop.

`running → awaiting_children → running`
Parked while its children work, holding no worker at all — until the last child's finish wakes it.

● THE WHOLE TRICK, IN ONE QUERY

```
-- Claim a batch of jobs atomically — any number of workers, no lock manager
WITH claimable AS (
    SELECT id FROM step_attempts
     WHERE status = 'queued'
       AND available_after ⩽ now()          -- backoff / not-before gate
       -- (+ per-tenant & global in-flight caps)
     ORDER BY priority, created_at
     LIMIT :batch
     FOR UPDATE SKIP LOCKED                 -- two workers never grab one row
)
UPDATE step_attempts s
   SET status = 'claimed',
       claimed_by = :worker,
       fencing_token = nextval('fencing_seq'),  -- new epoch: stale writers lose
       lease_expires_at = now() + :lease       -- renewed by heartbeat while owned
  FROM claimable
 WHERE s.id = claimable.id
RETURNING s.*;
```

That single statement is the entire coordination layer. No Redis, no lock service — Postgres row locks decide who gets what, and the lease column is the dead-man's switch a reclaim sweep watches.

§04 The parts that are actually hard

A durable queue is easy to sketch and unforgiving to run. A handful of races and failure modes decide whether it survives contact with production — and each has a small, boring answer that only looks obvious in hindsight.

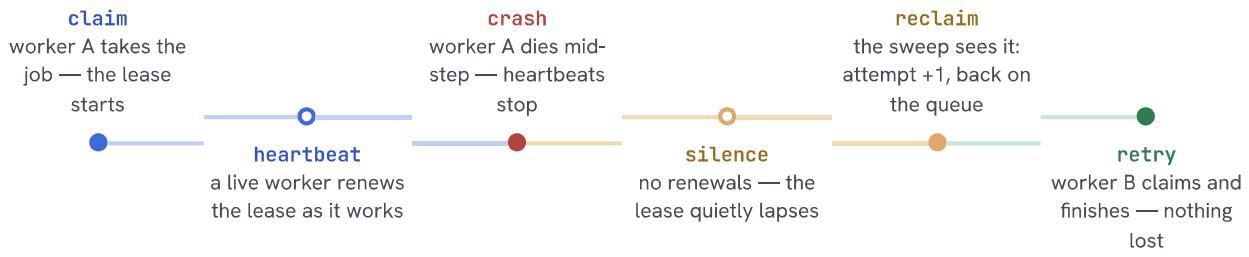
a Who advances the parent when the last child finishes?

Fan a step into 10,000 children and two can finish the same instant — both check "are we all done?", both see themselves as not-last, and the parent stalls forever. The fix is to make that check *idempotent and re-runnable*: finishing a child pokes the parent; the poke only asks "all children terminal?" and does nothing if not; and because a poke can be lost, a periodic sweep asks the same question on a timer. The parent advances exactly once — whether by the last poke or the next sweep.

Fig. 3 · Fan-out and the parent's sleep. The wake check is a question, not a counter — asking twice can't double-count.

b A five-minute step under a shorter lease

Set the lease too short and the reclaimer steals a step that's still running, so it executes twice; too long and a genuinely dead worker's job sits frozen. The answer is a **heartbeat**: a live worker keeps renewing its lease while it makes progress, so the lease itself can stay short. Renewal is guarded by ownership — a worker only renews rows it still holds — so if a row was reclaimed out from under a stalled worker, its late heartbeat quietly no-ops instead of fighting the new owner — in production, a 180-second lease renewed every 30 seconds. And a **fencing token** stamped at claim time is the backstop: a reclaimed row is re-claimed under a fresh token, so if a wedged worker ever wakes and tries to write its result, the write is rejected for carrying a stale one.



after three failed attempts → dead_letter — quarantined for a human, never an infinite loop

Fig. 4 · The lease is a dead-man's switch: the system never asks a dead worker to report its own death.

c Resuming without redoing the work

Here it helps to separate two records. **Save points** are the coarse, retryable rows the engine resumes from. Beside them runs an append-only **flight recorder** — every model call and tool result, written as it happens. A reclaimed agent doesn't restart from zero; it reads back its own flight recorder, rebuilds the state it had, and continues from the last completed step. The one call that was in flight when it died is marked *interrupted* — *verify before retrying* rather than blindly re-run, so a half-finished external effect gets checked, not duplicated.

d When the work itself kills the worker

A poison pill — an out-of-memory, an infinite loop — can kill the process before it records its own failure. That's fine, because the *reclaimer* owns the attempt count, not the worker: a dead worker's lease simply lapses, the reclaimer bumps the try count and re-queues, and after a fixed ceiling the row goes to `dead_letter` instead of looping forever. The same lease machinery that survives a crash also bounds a job that is fatal by nature.

THE TWO GUARDS, VERBATIM

```
-- Heartbeat: renew only what you still own
UPDATE step_attempts
   SET lease_expires_at = now() + '180 seconds'
 WHERE id = :step AND claimed_by = :worker;   -- no-ops if reclaimed (one-boot ids)

-- Fencing: a frozen worker's late result is rejected as stale
UPDATE step_attempts
   SET status = 'succeeded', result = :result
 WHERE id = :step AND fencing_token = :token;  -- a later claim holds a newer token
```

THE ENGINE KEEPS ITS OWN CLOCK

Scheduling and events, for free

An autonomous agent doesn't wait to be asked — it wakes itself, and both triggers we'd normally bolt on are already here. The claim already skips a row until its `available_after` time has passed — so stamping a future time *is* scheduling, and a row that re-enqueues itself with a fresh one is recurrence. Event-driven wakeups are just an inserted row a waiting step keys on. Delivery is the same fire-and-forget kick from Fig. 2, backed by a periodic sweep — so a dropped signal is never fatal, only a little slower, because the timer always asks again. And a long wait is free to hold: a step blocked on a human approval or a multi-day timer isn't a leased worker sitting idle — it's a parked row, durable and unleased, that the same due-time gate wakes when its moment comes.

§05 A step is a recipe — and that's the door

One property of this design is easy to miss, and it turns out to be the whole point. The worker never knows what a step *does*. Every step type is a recipe registered by name in one shared registry — fetch the data, transform the records, send the notice — and the engine claims, leases, retries, parks, wakes, and records all of them identically. It can't tell them apart, so it protects them all equally.

Which means anything you can express as a recipe inherits every guarantee in this part without asking. That is the property everything in the next part stands on.

PART IV

The agent

The move the whole paper was building to: mount the loop where durability already lives.

Here is why we built all of this. An AI agent's loop — *think, act, record, repeat* — is just one more recipe. Register it, and everything the engine guarantees snaps into place underneath every thought the agent has.

A · THE LOOP, MOUNTED

The harness becomes a property of the engine

Mechanically, "mounted" means *wrapped*. Whenever a worker runs a step — any step — it does the same three things around the recipe's code: **before**, claim the row and start the lease; **during**, keep the heartbeat going; **after**, write down how it ended — success recorded, or failure caught and re-queued. The wrapper is identical for every recipe and never looks inside it. The agent's entire loop — every thought, every tool call — runs within that envelope.

Seen this way, the *agent harness* — the scaffolding every team ends up building around their loop: retries, resumability, tracing, budgets — stops being a second system. It becomes a **property of the workflow engine** (Fig. 5). We didn't wire durability into the agent; we mounted the agent where durability already lives. Every one of those concerns, the agent gets by standing on the engine rather than carrying its own.

One tension is worth naming before a careful reader trips on it. This paper will argue, in Part V, that steps should be small — shrink what "twice" can touch — and an agent step is enormous: up to 25 iterations and 50 tool calls inside one lease. The resolution is that the agent step is **coarse on the outside and fine-grained on the inside**. Every iteration — the model's thought plus the tools it ran — becomes a durable row as it happens. So a recovered loop doesn't restart from zero: it rebuilds its transcript from those rows and resumes at the last completed iteration. A tool still in flight at the moment of the crash isn't blindly re-run — it comes back marked *interrupted, verify before retrying*, the Part V idempotency rule enforced at the loop's own boundary. The step is big; what a crash makes it redo is one iteration, not the whole run.

Two records make that possible: the engine *retries* from coarse save points — one per step — but *rebuilds* from a fine-grained flight recorder that logged every iteration. The save point is where it restarts; the recorder is what lets it restart without re-thinking.

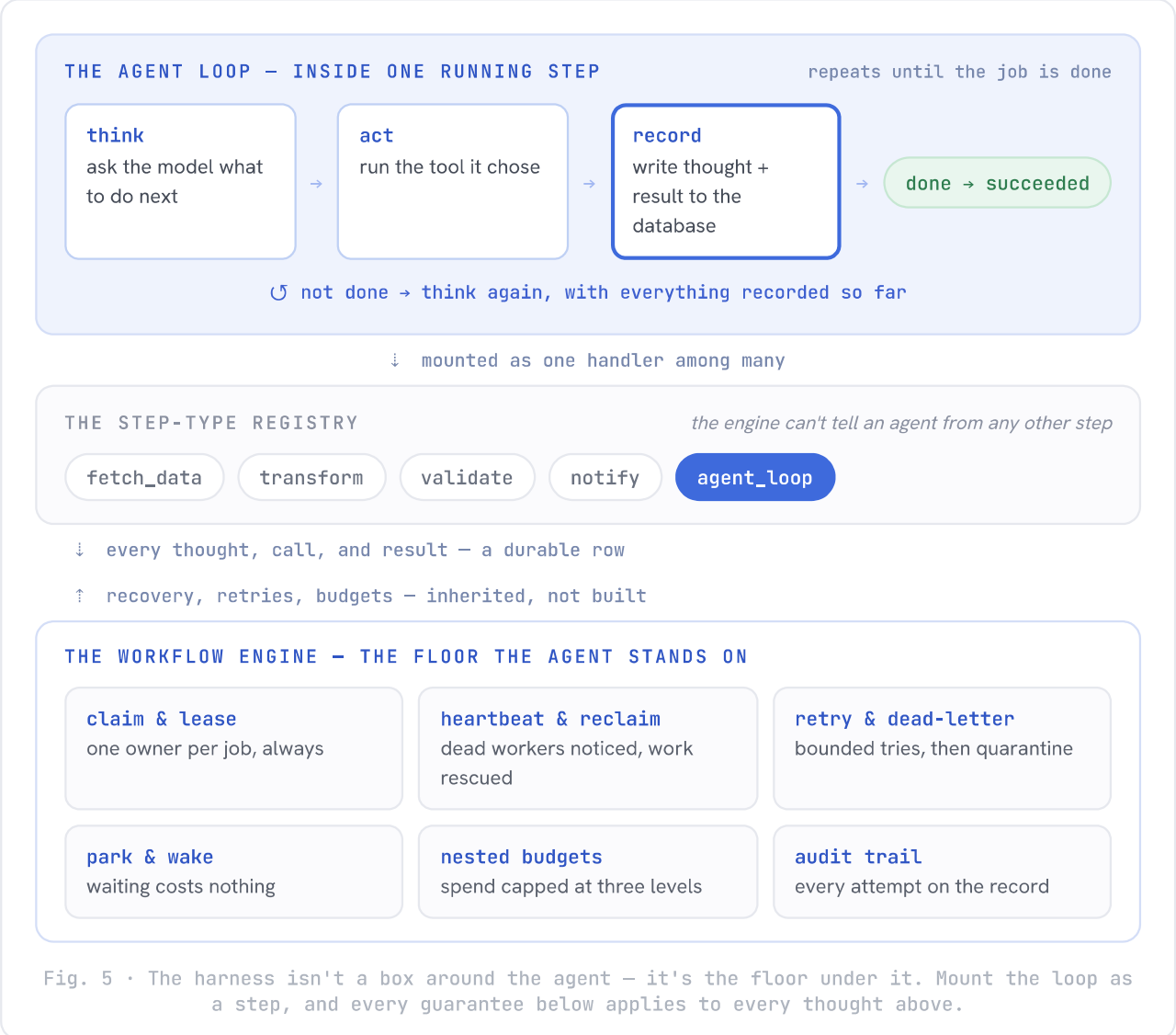


Fig. 5 · The harness isn't a box around the agent — it's the floor under it. Mount the loop as a step, and every guarantee below applies to every thought above.

B · A STEP THAT SPAWNS STEPS

Subagents: one job, past one context window

If one agent is a recipe, an agent that *delegates* is just a step that files more steps. A *sub*agent is a step that fans out child steps and waits on them (Fig. 3): the parent parks at `awaiting_children` and runs no code; when the children finish it's woken, rebuilds their results from the durable record, and continues. This is shipped, not aspirational — and it's how one job reaches past a single context window: each sub-agent works in its own.

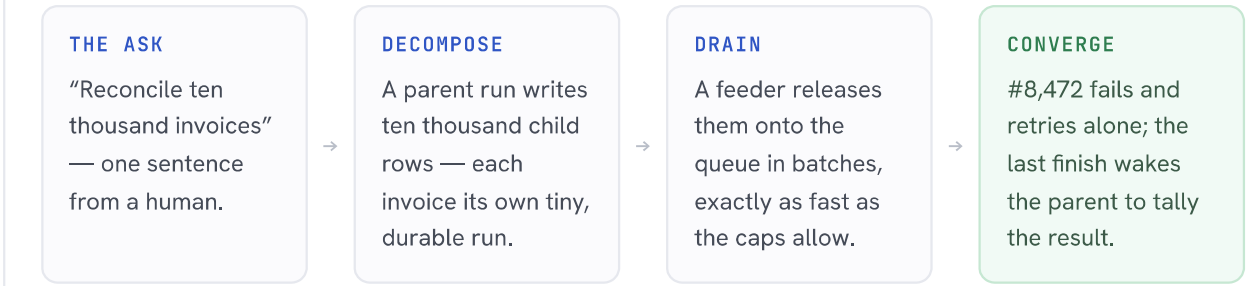


Fig. 6 · One ask becomes ten thousand durable runs, then one durable answer — Fig. 3's wake at full scale.

C · THE PLAN IS DATA

A dynamic DAG the model writes as it thinks

A classic workflow declares its graph before the first step runs. An agent can't — it discovers the shape of the job as it goes. So here the plan is not code waiting to execute; it's rows the model writes. When the loop decides a job needs three lookups and a reconciliation, it files them as child steps, and the graph materializes in the database one decision at a time, under the budgets below. Kill every worker mid-plan and the *plan* survives, not just the progress — because the plan was never in memory either.

And because every planned step is an ordinary row, each one arrives with three properties the harness never had to build:

retryable
bounded attempts, backoff, dead-letter — per planned step, not per job

replayable
re-run the job from any chosen step — current state is a read, not a memory

citeable
every attempt, thought, and cost on the record — the answer can point at the rows that produced it, which is the difference between an answer and an audit

This is also the deepest consequence of the choice back in Part II: a replay-based engine needs its planner to be deterministic, and a model is anything but. A state-based engine doesn't care. A nondeterministic planner writing a durable plan is exactly the combination the rest of this paper was built to make safe.

§06 What a runaway agent can't do

An agent that can spawn agents is, financially, a loop that can spend money — and durability makes that sharper, not softer: a resilient system will happily retry its way through your budget. So spend is bounded by **nested caps**, checked before every model call. One honest note on that phrasing: a check is a read followed by a spend, so many concurrent children can briefly overshoot a cap before the next check catches it — bounded by batch sizes, settled by the append-only cost ledger. The caps are circuit breakers, not accounting to the cent.

`per-workspace`

a daily budget across all of a customer's agents — the outer blast wall; hit it and new spend pauses until tomorrow — $50, in our deployment

`per-tree`

one agent plus every sub-agent it spawns, bounded together on cost and wall-clock — recursion can't compound into a fortune — $5 and 30 minutes, in ours

`per-run`

one agent's own loop — iterations and tool calls capped — 25 and 50 — so a single agent can't spin

The subtle one is the per-tree cap. A parent that spawns children goes to sleep — it runs no code while they work, yet their cost keeps accruing to its tree. So the cap can't be enforced by the sleeping parent alone; it's *also* checked by whatever stays awake to shepherd the children, reading the same cap definition. A sleeping parent can't overspend through its kids, because someone awake is watching the total.

PART V

The honest limits

Where the guarantees end — said plainly — and the levers that hold under real load.

§07 When a step runs twice

All this retrying raises the question a careful reader is already asking: *what if the step already did something before it died?* Slow the crash down and watch it. A worker's job is to push invoice #8,472 to the ERP, then write “done” in the database. It dies in between. The push happened; the “done” never landed. The database — telling the only truth it knows — still says *not done*, so another worker picks the row up and pushes the invoice again. The ERP just received it twice.

This is the one honest limit at the heart of the design — and no system can remove it without cooperation from the other side of the wire:

INSIDE POSTGRES

Exactly once

Row state, transitions, the event log. A transaction commits or it doesn't. The database guarantees it.

OUTSIDE THE BOUNDARY

At least once

The ERP call, the CRM write, the email. A crash mid-step means it may run twice. You must design for that.

So you design for the twice. **First, keep the damage small (Fig. 7).** Don't run “process ten thousand invoices” as one giant job — run ten thousand tiny ones, one invoice each. Now a crash touches exactly one invoice, and the retry re-does one push, not the whole batch. **Second, look before you leap on resume.** A replacement worker finds everything the first attempt already recorded and continues from there — and for the one call that was mid-flight when the worker died, it asks “did this actually go through?” before sending again.

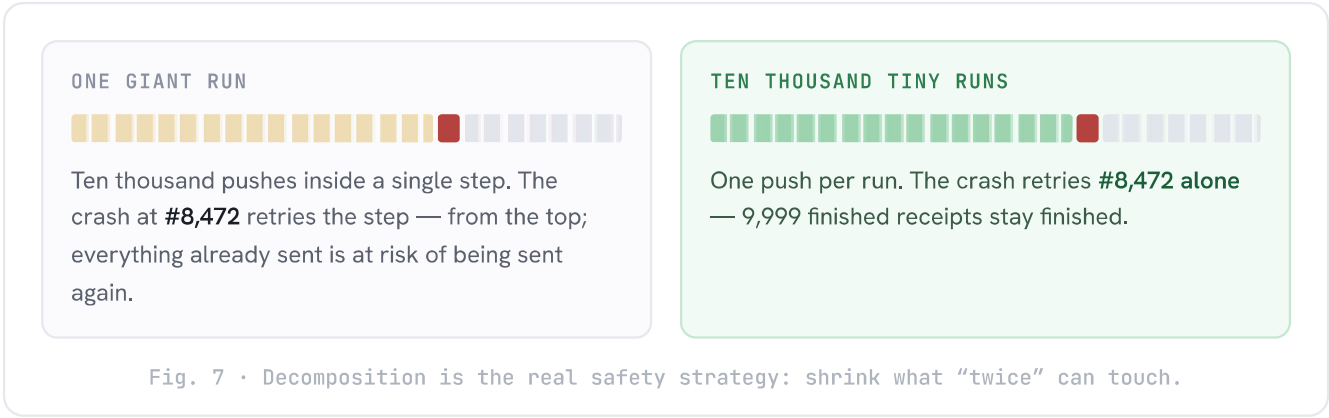


Fig. 7 · Decomposition is the real safety strategy: shrink what "twice" can touch.

Checkpoint every external effect back into the database the instant it succeeds. The database isn't only your queue — it's your *receipt*. Far from breaking the philosophy, this is the philosophy: the outside world is real, so record what you did to it, durably.

WHAT DURABILITY DOES NOT BUY YOU

Exactly-once stops at the database's edge. One rare race even the lease can't close: a worker *freezes* — not dead, just stuck — so the reclaimer hands its job to a second worker… and then the first wakes up. Both can call the ERP before either writes a word. The database stays correct (the frozen worker's late write is rejected as stale), but the outside world already heard the call twice. Model calls add a wrinkle of their own: run one twice and you pay twice, and the second answer won't match the first. So we don't claim the impossible — we keep the damage small and priced-in. Where the other system supports it, the call carries a **receipt number** (an *idempotency key*) so a duplicate is recognized and ignored. A resumed job checks before re-sending. Big jobs are split so a retry touches one item. And spend caps mean a duplicated model call costs cents, not correctness. Where the guarantee ends, we say so — rather than let "durable" quietly imply "safe."

§08 Holding up under load

"Just use Postgres" invites the obvious objection: does it hold up? For this class of work — thousands to hundreds of thousands of items, many tenants at once, runs measured in minutes and hours — yes, comfortably. Not because the pattern is magic, but because the pressure

points are known. The claim and the reclaim each run off a **partial index on exactly their predicate**, so neither ever scans the table; past that, three failure modes matter, and each has a named lever.

PRESSURE

Table growth

The firehose tables — events, traces, costs — grow far faster than the queue itself; unchecked they bloat and slow every scan.

Lever

Age-based retention sweeps purge the firehose on a timer, so the working set stays small and autovacuum keeps up.

PRESSURE

Connections

Postgres connections are heavyweight; thousands of workers can't each hold one open.

Lever

Put PgBouncer in front; workers share a small pool instead of one each.

PRESSURE

Fairness

One tenant's 200k-row migration will starve everyone else on a naive FIFO queue.

Lever

The claim enforces per-tenant and global in-flight caps — with a small, documented over-claim — so one run can't monopolize the workers.

WHERE THE CEILING ACTUALLY IS

This is not a million-events-per-second streaming bus. Every transition is a durable write — you are trading raw throughput for the guarantee that nothing is ever lost. For agentic back-office work that ceiling sits far above where the workload itself changes shape: you'll re-architect the *job* long before you outgrow the database.

THE TWO HOT-PATH INDEXES, AS DEPLOYED

```
-- the claim reads only this
CREATE INDEX idx_claim ON step_attempts (priority, available_after, created_at)
  WHERE status = 'queued';

-- the reclaim reads only this
CREATE INDEX idx_lease_expired ON step_attempts (lease_expires_at)
  WHERE status IN ('claimed', 'running');
```

THE ENVELOPE, CONCRETELY

What production actually enforces: 180-second leases renewed every 30; batches of two per worker tick; three attempts, then dead-letter; 25 iterations and 50 tool calls per run; $5 and 30 minutes per agent tree; $50 per workspace per day. And what we won't publish: a synthetic throughput benchmark. We haven't run one — the honest ceiling for this class of work is the envelope above, not an invented jobs-per-hour figure.

And the schema that carries it: claiming reads a partial index built on exactly its own predicate (`status = 'queued' AND available_after ≤ now()`), reclaiming another (`status = 'claimed' AND lease_expires_at < now()`) — so neither hot path ever scans the table — while the firehose tables are swept on age timers, traces keeping 90 days.

§09 The seams we accept

A layered, multi-table model has real costs, and naming them is part of the design. When one truth is spread across rows in two tables, they *can* drift — a step says succeeded while its run still says running. You pay a join to read an agent. A couple of registries can fall out of sync. None of it is free.

The discipline that keeps drift survivable is a **single chokepoint** per class of write: every state change of a given kind goes through one function, and a build-time check fails the build if any code writes around it. Invariants you enforce structurally outlast the people who remember them.

WHY ACCEPT THE SEAMS

Because the alternative was worse, and not hypothetical. A single fat table streaming every update crashed the browser with an event storm; inlining bulk payloads blew out memory. Splitting the firehose into its own tables and passing references instead of payloads engineered both failures away. A seam you can guard with a chokepoint beats an outage you can't.

PART VI

The horizon

What the same tables unlock next, and what’s worth stealing on your way out.

WHERE THIS POINTS

If a tool is a step, any system is reachable

A tool is already just a registered capability the engine calls as a step — so it already arrives wrapped in durability: retried on failure, resumable if the connection drops, its result checkpointed the instant it returns. The forward door follows directly. Speak an open tool standard like **MCP** and any compliant server — the ERP, the CRM, the ticketing system, the warehouse — plugs into that same step machinery and inherits the wrapper: retries, resumability, the checkpointed receipt. What it does *not* inherit is a pass on §07 — every new downstream still needs its own idempotency key, its own receipt, its own scopes. Durability wraps the call; it doesn't absolve the integration. We're not there yet; the point is that the architecture makes it a small step rather than a new system. That's the promise the whole design is pointed at: an agent that can touch anything in your company, and never lose its place doing it.

WHAT WE'RE WORKING ON NEXT

Three extensions the tables already permit

Verifier steps — quality control as more graph. A step is *done* when its transaction commits; whether it's *good* is a separate question — so make the check its own step, and make it a ladder. Deterministic checks first, because most of this domain is checkable by code: the sums reconcile, the counts match, every cited row exists. A cheap model judge only for what code can't check — relevance, reasoning — ruling against a written rubric. And a human gate (already first-class as `awaiting_approval`) for anything consequential. Every rung's verdict is a durable, citeable fact that ordinary dependency handling acts on; nothing ever un-succeeds. The registry makes this an afternoon, not a redesign.

Steering signals between agents. Today, siblings in a wave can't talk mid-flight — quarantine by design, so one hallucinating child can't poison thirty-nine others. When a real job shows the wasted work outweighing that protection, the extension is already sitting in the substrate: rows are messages, and every agent loop already reads state at each iteration boundary — a shared findings board and a cooperative wind-down flag slot in at that same checkpoint, with no new race machinery.

Published crash evidence. "Kill anything, lose nothing" deserves numbers, not prose. Next is a failure-injection suite — deliberate kills, stalls, and dropped wake-ups against an instrumented fake downstream — reporting the one metric that matters here: *duplicate external effects per thousand steps*, not completion rate. That table is the artifact we intend to publish after this one.

---

§10 The patterns worth stealing

01 **Postgres as the only state store.** Make compute disposable; your durability ceiling becomes the database's, not a process's uptime.

02 **Pick grains before tables.** Run, step-attempt, agent, and trace are four different questions. One table per grain.

03 **Atomic claim + lease + reclaim.** `SELECT … FOR UPDATE SKIP LOCKED`, a lease column, and a reclaim cron are a complete durable queue.

04 **Transitions update; retries append.** The live attempt moves through statuses, a retry files a new row, events log every change; collapse to “current” with status priority, not `MAX(attempt)`.

05 **Exactly-once state, at-least-once edges.** Trust the DB for your own state; shrink the blast radius so a retry re-does one item, not the whole job.

06 **An agent is a step type.** Plug the loop into the existing registry and inherit leases, retries, and recovery for free.

07 **Save points vs. flight recorder.** Coarse retryable rows to resume from; a fine append-only record to rebuild memory from. Recover at the save point, replay from the recorder.

08 **Decompose to beat the context window.** Many small runs with fresh windows + durable shared memory give a system unbounded reach from bounded agents. The skill is decomposition.

Put the truth somewhere that outlives the process, and long-running autonomous work stops being fragile.

It becomes what it always should have been — a very long sequence of small, durable, recoverable steps, running on a database you already trust.