DEEP DIVE

Building autonomous agents for regulated workflows

How we're building a long-running AI agent harness for FP&A and compliance-heavy workflows.

Verifiability · Controllability · Accountability

Most AI agent demos can't survive an audit.

Planning is converging. Execution is where every enterprise pilot dies.

The pattern that's failing

Every demo you've sat through. Every framework that promises "give the agent tools and let it figure it out." The same architecture, dressed in different libraries.

THE MONOLITHIC REACT LOOP

Give the LLM a goal. Give it tools. Let it loop.

The entire reasoning lives inside one rolling LLM conversation. "I should look up X, then check Y, then decide Z" — all internal context. The model is both the planner and the executor.

Fine for prototypes.
Fine for demos.
Fine for chatbots.

WHAT IT CAN'T DO

Survive an audit.

- × No audit trail — reasoning is internal context, not durable state
- × No intervention — humans can't gate or kill mid-loop
- × No attribution — every action runs under one anonymous identity
- × No recovery — if the agent dies mid-task, work is lost
- × No replay — the chain of thought evaporates the moment the loop ends

The demo gap and the production gap are the same gap. It's architectural, not aspirational.

Planning is solved. Execution isn't.

Two trends that nobody disagrees about — and one conclusion most are missing.

Frontier LLMs are getting great at PLANNING

GPT-5, Claude Opus 5, Gemini 3. "Given a goal, decompose it into steps" is becoming a solved problem inside 24 months.

Bottleneck shifts to EXECUTION

Reliability, observability, recovery, intervention, attribution. Everything around the LLM call. Where every enterprise pilot stalls.

The industry is converging on a specific answer

Anthropic's "Building effective AI agents" essay (Dec 2024) explicitly recommends workflow patterns under agents.

LangGraph · OpenAI Agents SDK · AWS Step Functions + Bedrock · Cloudflare Workflows + AI · Salesforce Agentforce · Microsoft Copilot Studio — all converge on the same primitive: **graph/DAG + dynamic dispatch + persistent state.**

DAG = directed acyclic graph. A list of tasks where each one says "I need these others to finish first." Steps and dependencies. That's it.

THE FRAMEWORK

Three things every enterprise AI agent must have

Without all three, an agent is a demo. With all three, it's deployable.

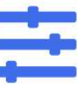
01

Verifiability

Every action is independently checkable, after the fact.

- Immutable audit trails
- Replayable execution
- Provenance of every input
- Citation of every decision

02



Controllability

Humans constrain, intervene, override — in real time.

- Permission scoping
- HITL approval gates
- Kill switches & thresholds
- Segregation of duties · rollback

03

Accountability

Clear, attributable responsibility for every outcome.

- Named service principals
- Liability mapping
- Regulatory attestability
- Conduct standards

Hold these three. Everything that follows traces back to one of them.

STAKES

Why this matters in FP&A and compliance

These aren't abstract concerns. Each pillar maps to a specific regulatory framework with material consequences.

SOX 404

Public-company financial controls

WHEN IT FAILS

Material weakness disclosure · CFO/CEO personal liability · stock impact

SR 11-7

Bank model risk management

WHEN IT FAILS

MRA findings · capital surcharge · examiner enforcement

MiFID II

EU investment firm conduct

WHEN IT FAILS

Best-execution failures · client suitability breaches · fines

EU AI Act

High-risk AI system classification

WHEN IT FAILS

Conformity assessment · post-market monitoring · 7% revenue fines

Each of these regimes asks the same questions of any system that touches material decisions: **Can you prove what it did? Can you stop it? Who owns the outcome?**

An LLM in a chat window cannot answer any of these. An agent with the right architecture can answer all three.

PILLAR 01

Verifiability

Every action and output must be independently checkable after the fact. Months — or years — later.

Auditable

Every state change of every step writes an immutable row. The audit log is the system of record — not a debug feature, not optional logging. It's the same artifact regulators inspect.

Replayable

Every step's inputs and outputs are stored. Re-execute any step against its captured inputs. LLM calls are stochastic, but the surrounding system is deterministic — same inputs yield logically equivalent outputs, and cached responses give bit-perfect replay.

Explainable

The DAG itself is the chain of thought. You don't ask the agent why it did something — you read the graph. Reasoning is externalized at every step boundary, not hidden inside a context window.

WHAT THIS REQUIRES ARCHITECTURALLY

Externalized state on every step boundary. An immutable event log. A graph structure that captures the agent's reasoning, not just its outputs.

PILLAR 02

Controllability

Humans must be able to constrain, intervene in, and override the agent — in real time.

Permission scoping

Agent only touches what it's been granted access to. Scope is checked at execution, not at trust.

HITL approval gates

Material actions pause for human review. Run can park indefinitely without losing state.

Kill switches

Cancel a run mid-flight. All in-flight jobs flip to cancelled. No orphans.

Thresholds

Dollar limits, transaction caps, rate limits per skill. Hard cutoffs the agent cannot cross.

Segregation of duties

Same agent cannot both initiate and approve. Roles enforced at the step level.

Rollback

Reverse the last N actions. The audit log makes this surgical, not destructive.

WHAT THIS REQUIRES ARCHITECTURALLY

Discrete steps the system can pause, gate, or kill independently. State machines with terminal states. Cascading cancellation. Hooks at every boundary.

PILLAR 03

Accountability

Clear, attributable responsibility for every outcome — when things go right, and when they don't.

Named service principals

The agent acts under a specific identity, not anonymously. Every action is attributable to a workspace, a workflow, a step, a skill.

Liability mapping

Each class of decision has a designated human or entity responsible for it. The agent's authority is bounded; the human's accountability is explicit.

Regulatory attestability

A compliance officer can sign off that the agent meets SOX/SR 11-7/MiFID II requirements. The audit trail and control framework support that signature.

Conduct standards

The agent's behavior is evaluated against the same fiduciary, suitability, and fairness rules as a human employee performing the same role.

WHAT THIS REQUIRES ARCHITECTURALLY

Multi-tenant identity on every row. Bounded authority per skill. Auditable boundaries between agent decisions and human approvals.

Three pillars trace back to two architectural properties

Verifiability, controllability, accountability — all three are downstream of two design choices most agent systems skip.

PROPERTY 01

Dynamic DAGs with completion guarantees

The graph isn't fixed. The agent designs it — and mutates it as sub-agents discover work. But every safety property holds across the mutation: Kahn's algorithm runs against whatever shape exists right now. Cascade-cancel works. Lease reclamation works. Dead-letter works.

The agent gets imagination. The engine gets discipline.

PROPERTY 02

Stateless workers, stateful Postgres

Every actor in the system except the database holds zero durable state. Workers can die, restart, scale, or get killed mid-execution. Recovery is automatic because the only memory that matters lives in Postgres — externalized at every step boundary.

The audit trail is the chain of thought. They are the same artifact.

MENTAL MODEL

FedEx solved this decades ago

FedEx solved the hard parts of high-volume, high-reliability execution decades ago. The architecture maps cleanly.

FEDEX		OUR HARNESS
A package enters the system	→	A workflow run is created
The shipping label	→	The DAG definition
Sortation hub assigns routes	→	The executor enqueues jobs
Drivers pick up packages atomically	→	Workers claim jobs via SKIP LOCKED
Scan events at every checkpoint	→	workflow_job_events — the audit log
Driver radio-checks back to dispatch	→	Heartbeats extend the lease
Lost package → exception handling	→	Failed job → dead-letter queue
Signature for high-value deliveries	→	HITL approval for material actions

Same primitives. Different cargo. Theirs is packages. Ours is agent decisions.

THE CAST

Three actors, one source of truth

The whole engine is three things talking to each other. Two of them are stateless — only Postgres remembers anything.

workflow-executor

The dispatcher

- Reads the DAG (possibly agent-generated)
- Decides what runs next via Kahn's algorithm
- Inserts rows into workflow_runs and workflow_jobs
- Never calls a skill itself
- Returns in <1 second on the chunked-fanout path

workflow-job-worker

The executor

- Claims jobs atomically — no two workers grab the same one
- Calls the skill — or runs the agent loop
- Heartbeats while running, lease auto-recovers if dead
- Reports outcome on success or failure
- Pings the dispatcher when its batch is done

Postgres

The single source of truth

- Holds every row of state
- Provides atomic claims for free
- Runs cron-based safety nets
- Survives any worker crash
- No Redis. No Kafka. No Temporal.

Kill any process at any moment — recovery is automatic, because state lives in the database, not in any worker's memory.

What the executor actually does

Verifiability hinges on this — every decision the dispatcher makes writes a row before any work happens.

1. **Read the DAG**

   Loads the graph — agent-generated or hand-written. Runs a topological sort to find which steps are ready (zero unfinished dependencies).

2. **Write the work down**

   For each ready step, inserts rows into the job queue — one per iteration in a fan-out. Each row has its own payload.

3. **Fire-and-forget**

   Sends a non-blocking ping to wake a worker, then returns. The caller gets a response in under a second, regardless of how many jobs were enqueued.

4. **Advance on completion**

   When workers finish a batch, they ping back. The executor checks whether all jobs for this step are terminal. If yes, aggregate and unlock downstream.

THE KEY INSIGHT

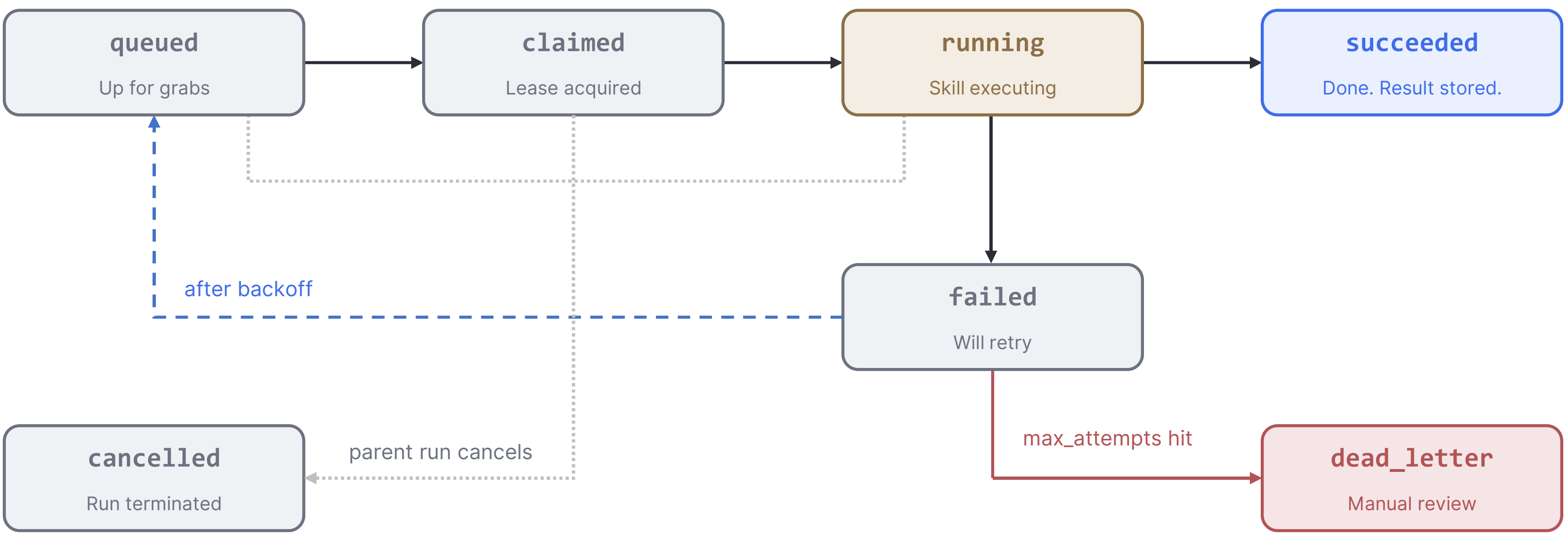
The executor is a dispatcher, not a worker.

It never calls a skill. It never waits for work to finish. It translates the DAG into rows in Postgres and walks away.

This is why one request can kick off thousands of parallel jobs and still return in under a second — the executor isn't doing the work, it's writing the work down.

Controllability at the row level

Every step is a state machine. The transitions are how the system enforces governance — pause, gate, kill, requeue, all visible in the status column.



`awaiting_approval` (HITL gate) and `cancelled` (cascade-cancel) are the controllability primitives. The cascade-cancel trigger flips queued/claimed jobs to cancelled when the parent run terminates.

The agent generates the DAG

A planning agent receives a natural-language goal and produces a workflow definition. The graph is no longer hand-written — it's emitted.

USER GOAL

"Reconcile this month's invoices to contracts, push to QuickBooks, flag anomalies, send me a report."

MASTER PLANNING AGENT

Decomposes the goal. Inspects available skills + connected systems. Emits a workflow_definitions row.

WHAT THE PLANNER EMITS

DAG shape

Steps and dependencies. Can be a single ForEach or a deeply nested graph.

Skill bindings

Each step is bound to a registered skill — bounds what can be executed.

HITL gates

Where to pause for human approval. Material actions never auto-execute.

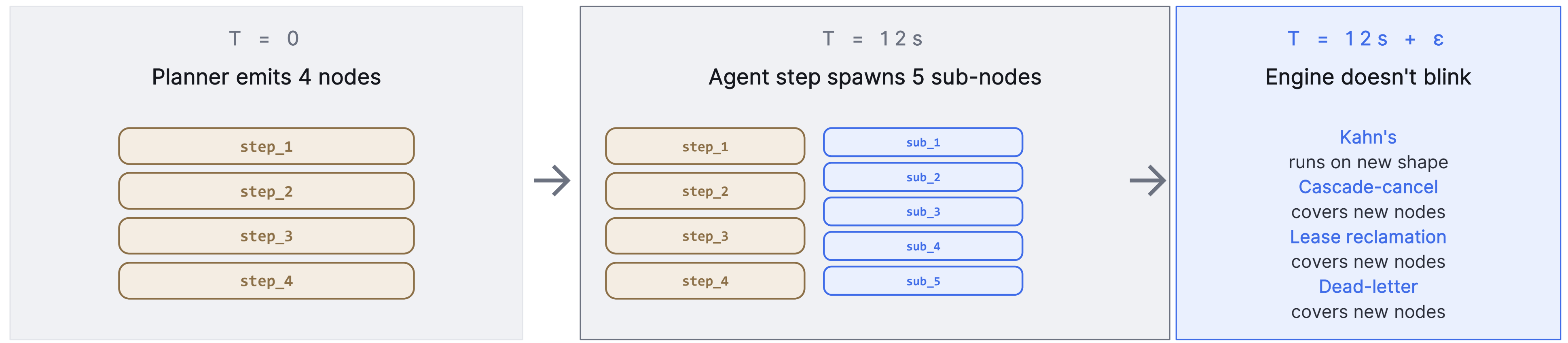
Scope & limits

Per-step max_iterations, max_cost_usd, max_attempts. Hard ceilings.

The DAG isn't fixed. It's the agent's plan, written down — and from this point on, the engine takes over.

Dynamic DAGs — the graph mutates as the agent runs

Agents discover work as they go. New rows get inserted into workflow_jobs at runtime. The engine doesn't care that they weren't there 10 seconds ago — it runs Kahn's against whatever shape exists right now.



THE COMPLETION GUARANTEE

The agent gets imagination. The engine gets discipline.

No special-case code for runtime mutation. The same primitives that handle a static 5-node DAG handle a 5,000-node graph that wasn't there 30 seconds ago.

Watch the DAG grow

A real workflow isn't a static graph — it discovers itself. Here's an invoice reconciliation run, with the DAG mutating as the agent finds work.

T = 0s

Planner emits the DAG

Node	Status
fetch_invoices	queued
extract_fields	queued
match_contracts	queued
post_to_qb	queued
send_report	queued

5 nodes. Linear plan.

T = 30s

Anomalies discovered

Node	Status
fetch_invoices	succeeded
extract_fields	succeeded
match_contracts	running
investigate_a	queued
investigate_b	queued
investigate_c	queued

Match step finds 3 mismatches.
Agent spawns 3 investigators.

T = 2m

Sub-agent spawns more

Node	Status
fetch_invoices	succeeded
extract_fields	succeeded
match_contracts	succeeded
investigate_a	succeeded
investigate_b	running
compare_rev_b	running
investigate_c	succeeded

investigate_b spawns its own comparison agent.

T = 5m

HITL gate, then close

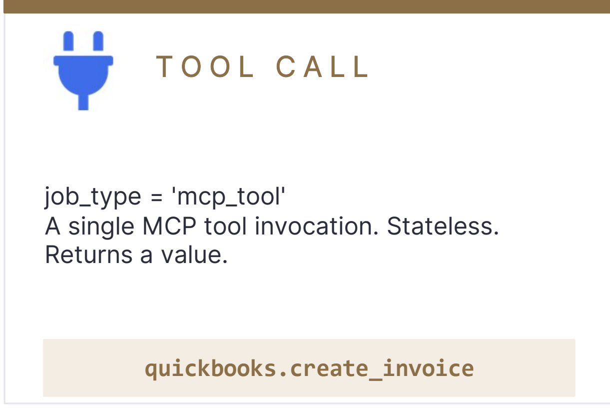
Node	Status
(8 prior succeeded)	
review_findings	awaiting_hitl
post_to_qb	queued
send_report	queued

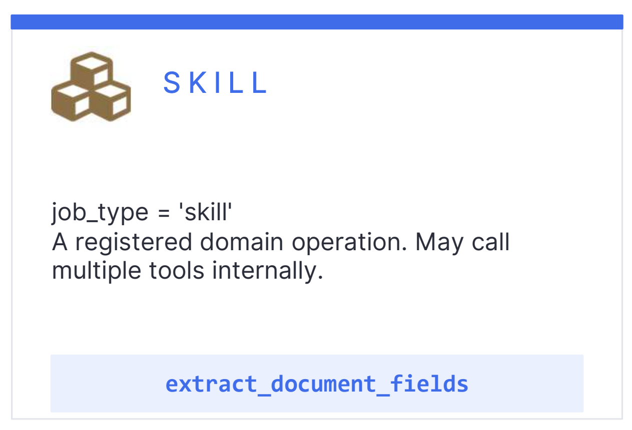
Material action paused.
Human reviews. Then resume.

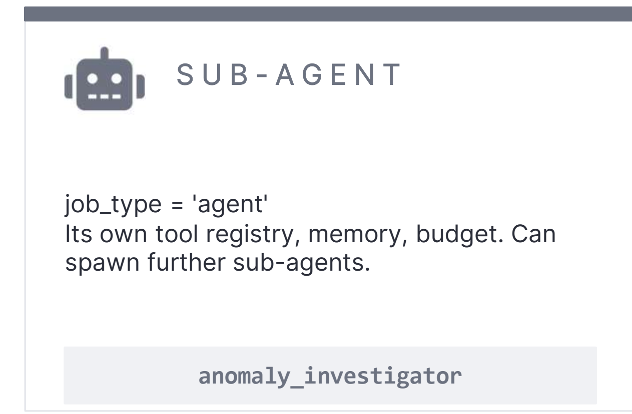
Same engine. Same primitives. The graph went from 5 nodes to 12 — and the engine never blinked.

Tools, jobs, and sub-agents are the same primitive

From the engine's perspective, calling a skill, running a sub-agent, and invoking an MCP tool are indistinguishable. They're all rows in workflow_jobs with different job_type values. This is why thousands of sub-agents in parallel costs nothing extra to support — it's the same SKIP LOCKED loop running across more rows.







ALL THREE ARE A ROW IN

`workflow_jobs`

Same SKIP LOCKED claim. Same lease + heartbeat. Same retry. Same audit trail. One primitive. Recursive composition. Horizontally scales to thousands of parallel sub-agents.

Why statelessness is load-bearing

Every governance property in this deck traces back to one design choice: workers hold no state, only Postgres does. Pull that out and the rest collapses.

MONOLITHIC REACT LOOP

Reasoning lives in volatile context

- × Can't replay — no record of intermediate decisions
- × Can't intervene — nothing to grab between steps
- × Can't attribute — no durable identity per action
- × Can't recover — agent dies, work is lost

DAG-STRUCTURED EXECUTION

Reasoning crosses Postgres at every step

- ✓ Replayable — every input + output captured
- ✓ Interruptible — pause, gate, cancel between steps
- ✓ Attributable — workspace_id + claimed_by on every row
- ✓ Recoverable — kill any worker, lease expires, requeue

THE INSIGHT

The audit trail is the chain of thought.

Monolithic agents have a hidden segment between writes — a black box you can't inspect. DAG-structured execution forces every reasoning step to externalize. The database is the audit trail; the audit trail is the chain of thought; they are the same artifact. That's why stateless workers + stateful Postgres is not a performance choice — it is the precondition for governance.

TL;DR

Three pillars. Two architectural properties. One thesis.

3 PILLARS

Verifiability · Controllability · Accountability — the irreducible requirements for any AI agent in a regulated workflow.

2 PROPERTIES

Dynamic DAGs with completion guarantees. Stateless workers, stateful Postgres. Together they make the three pillars achievable.

1 THESIS

The audit trail is the chain of thought. Externalize every reasoning step into Postgres and governance becomes a structural property — not a feature.

Building this for FP&A or compliance? DM me.