

WHAT IS AN AI AGENT HARNESS?

THE SEVEN LAYERS OF A PRODUCTION AGENT
ARCHITECTURE, AND WHERE THE HARNESS FITS

2026

THE HOT TERM OF Q4 2026

An **AI agent harness** is the software that turns a model into an agent. It runs the loop of calling the model, executing the tools it asks for and feeding the results back, while managing context, memory, permissions and session state.

It sits between whatever invokes the agent (a person, an app or a workflow) and the model provider, and it hands code execution off to a sandbox. Examples include **Claude Code**, **Codex**, **Orchestra’s Agent Runtime** and many others.

As a software company we had our own reckoning with AI, and it was about agents. Where agents run, and how they need to run, was not obvious. One thing was:

YOU EITHER DIE NOW AS A SOFTWARE COMPANY, OR LIVE LONG ENOUGH TO BUILD A CUSTOM HARNESS.

That means living long enough to work out what a harness actually is, then building one into your product, because every customer now expects an agentic experience. If you don’t, that’s fine. Just make sure your entire platform works via API.

Most of the confusion comes from mashing several layers together. In this paper we separate them into seven, from the top of the stack to the bottom:

- 1

The user surface (or trigger)

2

Orchestration and workflows

3

The AI agent harness

4

Tools and integrations

5

LLM hosting and inference

6

Compute and the sandbox

7

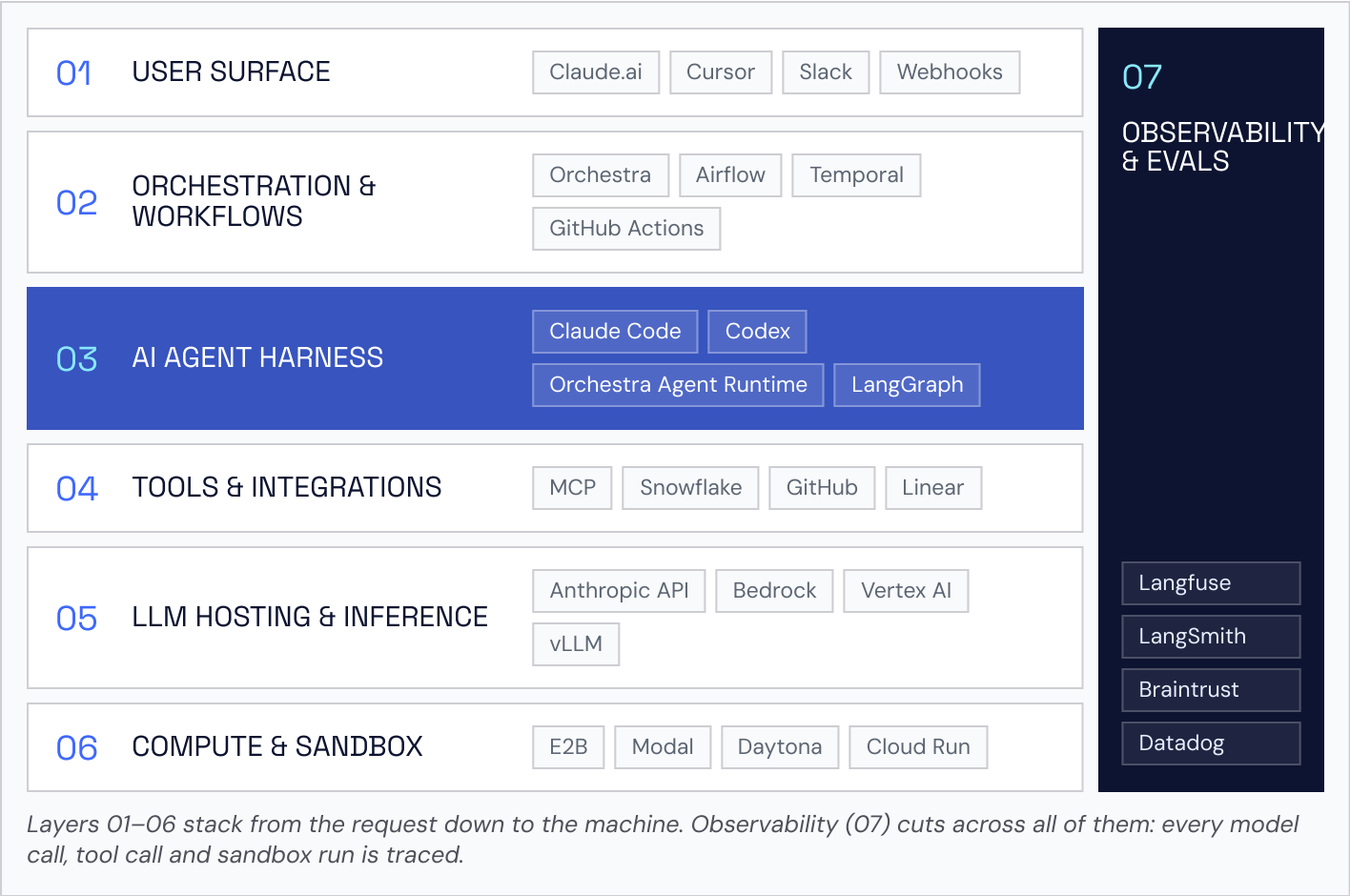
Observability and evals

THE THESIS

The agent loop itself is a commodity. The differentiation lives in the **skills, tools, permissions and context** you give it, and in the **control plane** that triggers, governs and pays for every agent you run.

THE SEVEN LAYERS OF AN AGENT ARCHITECTURE

The harness is one layer in a bigger system. Each layer has a different job, a different set of vendors and a different failure mode.



Read the stack top to bottom as the path of a single request. Something invokes the agent (01), possibly on a schedule or event (02). The harness runs the loop (03), calls tools (04) and models (05), and hands untrusted code to a sandbox (06). Everything along the way is recorded (07).

The rest of this paper takes each layer in turn with real examples, then puts them back together using a real incident: a failing dbt model, triaged by an agent overnight.

THE USER SURFACE

The surface is whatever the request comes in through: a chat window like Claude.ai, an IDE like Cursor, a terminal running Claude Code, or a side panel inside a SaaS product.

The architectural point is that **the surface should be thin**. It renders messages, streams tokens, shows tool calls and asks for approvals. It should not contain agent logic. If your “agent” only works inside your React app, you’ve built a feature, and you’ll rebuild it the moment a customer asks for the same thing in Slack.

The surface is often not a human at all. A GitHub webhook, a failed pipeline run or an inbound email can all start a session. The architecture shouldn’t change either way.

CHAT.JSX

THE WRONG WAY: THE AGENT LIVES IN THE BROWSER

```
import Anthropic from "@anthropic-ai/sdk";

// WRONG: the API key ships to every user's browser
const client = new Anthropic({
  apiKey: import.meta.env.VITE_ANTHROPIC_KEY,
  dangerouslyAllowBrowser: true,
});

export function Chat() {
  const [messages, setMessages] = useState([]);
  async function send(text) {
    const history = [...messages, { role: "user", content: text }];
    while (true) { // WRONG: the agent loop runs inside a React component
      const res = await client.messages.create({
        model: "claude-sonnet-5-5",
        system: "You are Orchestra's assistant...", // WRONG: prompt is in the JS bundle
        tools: TOOLS, messages: history,
      });
      history.push({ role: "assistant", content: res.content });
      if (res.stop_reason !== "tool_use") break;
      for (const block of res.content.filter((b) => b.type === "tool_use")) {
        if (!confirm(`Allow ${block.name}?`)) continue; // WRONG: permissions are a popup
        // ...call the API, push tool_result back into history
      }
    }
    setMessages(history); // WRONG: close the tab and the session is gone
  }
  return <ChatWindow messages={messages} onSend={send} />;
}
```

Five mistakes in one component: leaked key, loop in the UI, prompt in the bundle, permissions as a browser dialog, and no durable session.

A THIN SURFACE TALKING TO A HARNESS

The right version sends messages, renders events and relays approvals. Nothing else. The harness API below is illustrative.

CHAT.JSX

THE RIGHT WAY

```
export function Chat({ sessionId }) {
  const [events, setEvents] = useState([]);

  useEffect(() => {
    const stream = new EventSource(`/api/sessions/${sessionId}/events`);
    stream.onmessage = (e) => setEvents((prev) => [...prev, JSON.parse(e.data)]);
    return () => stream.close();
  }, [sessionId]);

  const post = (path, body) =>
    fetch(`/api/sessions/${sessionId}${path}`, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(body),
    });

  return (
    <ChatWindow onSend={({text}) => post("/messages", { text })}>
      {events.map((e) => {
        switch (e.type) {
          case "text":      return <Message key={e.id} text={e.text} />;
          case "tool_call": return <ToolCall key={e.id} name={e.name} status={e.status} />;
          case "approval_request":
            return <Approve key={e.id} summary={e.summary}
              onDecide={(allow) => post(`/approvals/${e.id}`, { allow })} />;
        }
      })}
    </ChatWindow>
  );
}
```

The payoff shows up when a customer asks for the same agent in Slack. With a thin surface it's a few lines against the same session API, because the harness already holds everything else.

SLACK_BOT.PY

SAME AGENT, NEW SURFACE

```
@slack.event("app_mention")
def on_mention(event, say):
    session = harness.sessions.get_or_create(thread_id=event["thread_ts"])
    session.send(event["text"])
    for e in session.events(until="idle"):
        if e.type == "text":
            say(e.text, thread_ts=event["thread_ts"])
```

ORCHESTRATION & AGENTIC WORKFLOWS

The most valuable agent work is **headless**, with nobody watching. It runs at 6am, when something breaks, or when another agent hands it a job.

The model labs have noticed. Anthropic added **Routines** to Claude, Cursor has **Cursor Workflows** and OpenAI has **Scheduled Tasks**. Today they share four problems:

- 1 **Weak orchestration.** Claude Routines run hourly, daily or weekly. That is a timer, not a scheduler.
- 2 **No context.** You wire up several MCPs to give the agent context, which is slow, expensive and hard to repeat.
- 3 **Not centrally managed.** They run locally, with no single pane of glass to govern and coordinate them.
- 4 **Expensive and tied to one model provider.** Nuff said.

Other approaches each solve part of the problem:

SIMPLE CRON & GITHUB ACTIONS Running Claude Code on every pull request is the canonical example. It works, until you have more than a handful of jobs.	DURABLE TEMPORAL & INNGEST Excellent for long-running, multi-step agent workflows that must survive restarts.	DATA-AWARE AIRFLOW, DAGSTER, ORCHESTRA Already know your pipelines, assets and failures, which makes them a natural trigger for agents that act on data.
--	---	--

Agents don’t replace orchestration. They increase it. Every agent is one more thing that needs triggering, sequencing, alerting and cost tracking. Multi-agent systems, where a planner hands work to specialists, are just workflows with more expensive steps. That is why a harness must be triggerable, pollable and cancellable via API.



HOW ORCHESTRA HELPS

Orchestra is a **serverless control plane** for data and AI workflows. The same platform that runs your pipelines triggers agents on any event or schedule, passes them the run context, and tracks what they did and **what it cost**, from one pane of glass.

THE HARNESS: THE LOOP AT THE CENTRE

Strip away the branding and every harness has the same core loop.

HARNESS.PYTHE MINIMUM VIABLE HARNESS

```
import anthropic

client = anthropic.Anthropic()
messages = [{"role": "user", "content": task}]

while True:
    response = client.messages.create(
        model="claude-sonnet-5-5",
        max_tokens=4096,
        system=SYSTEM_PROMPT,
        tools=TOOLS,
        messages=messages,
    )
    messages.append({"role": "assistant", "content": response.content})
    if response.stop_reason != "tool_use":
        break # the model thinks it's done

results = []
for block in response.content:
    if block.type == "tool_use":
        check_permission(block) # can it do this?
        output = run_tool(block.name, block.input)
        results.append({"type": "tool_result",
                        "tool_use_id": block.id,
                        "content": output})
    messages.append({"role": "user", "content": results})
```

This could run on an EC2 machine or a Lambda, probably wrapped in a backend like FastAPI. A production harness also has to handle:

CONTEXT MANAGEMENT Compaction and summarisation when long sessions blow through the window.	MEMORY What persists across sessions, and what doesn't.	PROMPTS & SKILLS Packaged instructions loaded only when relevant, not a 40-page prompt.
PERMISSIONS Tools that run freely, tools that need approval, and tools that never run.	HOOKS Inject logging, guardrails or cost limits before and after each step.	SESSION STATE Pausable, resumable and inspectable, even an hour in.

THREE CAMPS OF HARNESS

This is the layer with the most confusion about what is what. The options fall into three camps.

CAMP	EXAMPLES	WHAT YOU GET
Off-the-shelf	Claude Code, OpenAI Codex, opencode, Pi, Goose	A complete product you run locally or in CI.
Agent SDKs	Claude Agent SDK, OpenAI Agents SDK, LangGraph, Pydantic AI	The loop as a library. You shape the rest.
Managed	Claude Managed Agents, Orchestra Agent Runtime	The vendor runs the loop, tools and session state. Code runs in a sandbox you control.

Very few companies should hand-roll the loop in 2026. The interesting work is in the system prompt, skills, tools and permissions, because that is where your domain knowledge lives. Data teams should prefer a data-specific harness like the Orchestra Agent Runtime.

Interlude: desktop apps are back

Claude, Codex and Cursor all ship desktop apps, and Snowflake now has **CoCo Desktop**. A browser tab is good for chat and poor for work: it can't read the dbt project on your laptop, run a terminal command or use cached credentials. A desktop app is still just a surface. The question is where its harness lives.

ARRANGEMENT 1 HARNESS RUNS LOCALLY Claude Code in the desktop app, or CoCo Desktop running SQL with a locally cached OAuth token. + Credentials stay on the machine. Low latency, trivial setup. - Dies when the lid closes. Nothing triggers it at 6am. No central audit trail.	ARRANGEMENT 2 CLOUD HARNESS, DESKTOP BRIDGE Loop, state and history live on a server. The app exposes local folders and tools back to it. + Keeps working when you walk away. Local reach only when needed. - Harder to build. The bridge must be explicit about what it exposes.
--	---

The trap is the React mistake again with a bigger download. Most of these apps are built on Electron, and it is tempting to put the loop inside. The desktop app should render, approve and expose local tools. The harness owns everything else.

TOOLS, MCP & THE AGENT'S HANDS

A harness without tools is a chatbot with ambitions. The tools layer is everything the agent can actually do: query a warehouse, read a ticket, trigger a pipeline, send a message.

Sharing tools is hard, and agents need central access to them. And tools aren't just APIs: some APIs are massive, and an agent should only see the parts it needs. If a nested-JSON API you hate suddenly gets an MCP server, your agent can now burn credits paging through it.

The Model Context Protocol (MCP) is now the standard. A vendor exposes an MCP server and any compatible harness can discover and call its tools. Snowflake, GitHub, Linear, Slack, Attio, Figma and Orchestra all have one, so an agent in Claude Code can list failing pipelines, read task logs and rerun a pipeline with no glue code.

Tool design matters more than people expect. Exposing all 200 REST endpoints usually makes agents worse, because the model wastes context choosing between near-duplicates. **Fewer, well-named, task-shaped tools** with typed inputs and outputs beat a mirror of your API.

TOOLS.PY

ONE TASK-SHAPED TOOL, NOT 200 ENDPOINTS

```
from pydantic import BaseModel

class RerunFailedTasks(BaseModel):
    """Rerun only the failed tasks of a pipeline run. Use after a fix is merged."""
    pipeline_run_id: str
    dry_run: bool = True          # default to safe; harness asks for approval to flip

class RerunResult(BaseModel):
    rerun_id: str
    tasks_queued: list[str]
    estimated_cost_usd: float    # typed output the model can reason about
```

If you don't build a harness, your survival depends on being an excellent tool inside somebody else's. That is a fine strategy, but it raises the bar on your API. Agents punish bad APIs faster than humans do, because they can't read between the lines of your docs.

Finally, **private networks**. Most enterprise data isn't on the public internet, so the agent needs to reach internal MCP servers without punching holes in the firewall. Anthropic's MCP tunnels are one answer. Running the harness inside your own network is the other.

LLM HOSTING & INFERENCE

The layer everyone thinks about first, and the one that should take the least of your design energy.

OPTION	EXAMPLES	WHY CHOOSE IT
Model lab direct	Anthropic API, OpenAI API	Newest models first, plus prompt caching and batch processing.
Hyperscaler	AWS Bedrock, Google Vertex AI, Microsoft Foundry	Spend counts against an existing cloud commitment, and data stays inside an approved boundary.
Self-hosted	Open-weights models on vLLM	Very high volume, strict residency, or narrow tasks a small fine-tuned model can handle. Expect a GPU bill and an on-call rota.
Inference providers	Open-weights hosts	Useful for open models, but their costs track GPUs plus lab pricing.

For large enterprises the hyperscaler is usually the deciding factor. Half the battle in enterprise AI is getting the model onto a bill someone has already signed. On top sit gateways like **OpenRouter** and **LiteLLM**, which give one API across providers with fallbacks and spend tracking.

The principle: **the harness should swap models without a rewrite**. A cheap, fast model can classify an incident and a frontier model can write the fix. Model routing is only a cost lever if your harness doesn't hard-code one provider.

ROUTING.PY

DIFFERENT STEPS, DIFFERENT MODELS

```
import litellm

ROUTES = {
    "classify": "bedrock/anthropic.claude-haiku", # cheap, fast
    "fix":      "bedrock/anthropic.claude-opus",  # frontier
}

def call(step, messages, **kw):
    return litellm.completion(model=ROUTES[step], messages=messages,
                              fallbacks=["anthropic/claude-sonnet-5-5"], **kw)
```

RULE OF THUMB

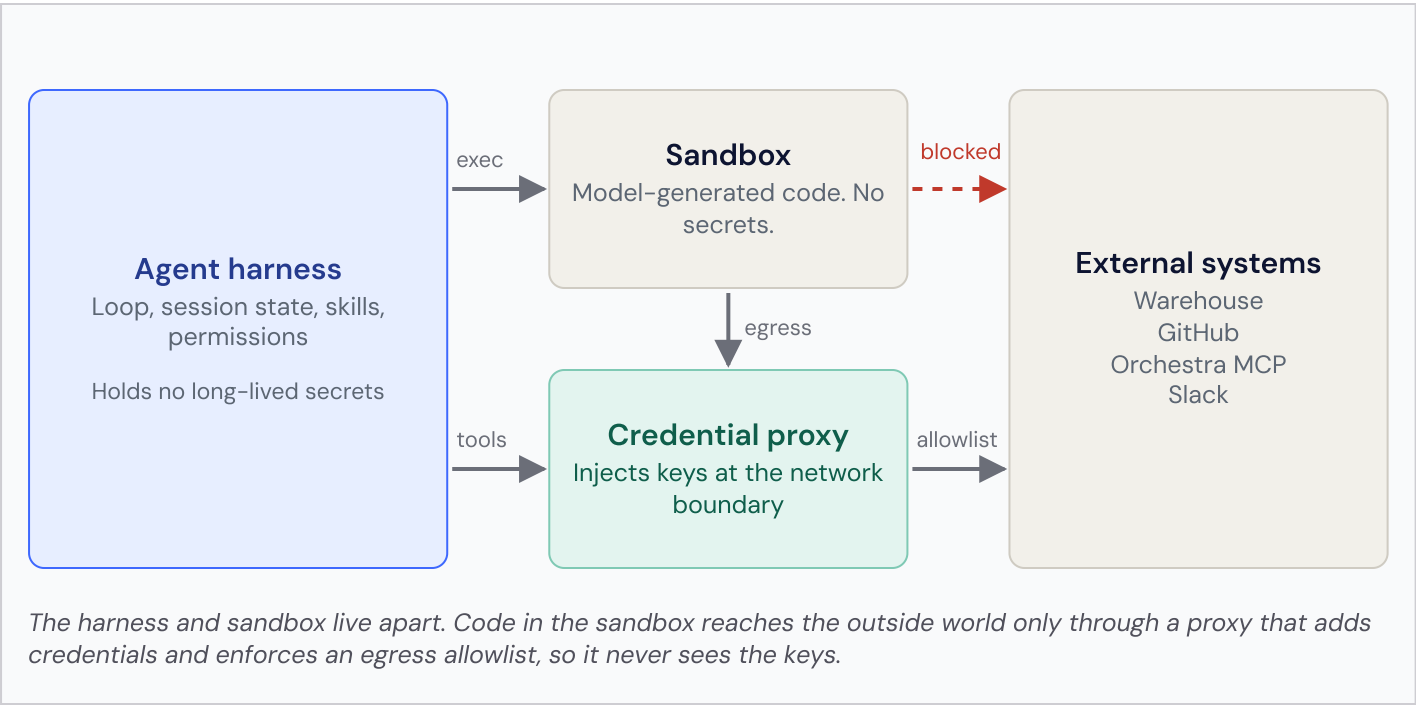
Pick inference on **procurement and data boundary**, not benchmarks. Design the harness so the model is a parameter.

COMPUTE & THE AGENT SANDBOX

Two separate questions: **where the harness runs**, and **where the code the agent writes runs**.

Agents are bursty and long-lived. A session might idle waiting for approval, then do ten minutes of intense work. AWS Lambda caps out at fifteen minutes, fine for a tool call but tight for a session. Cloud Run, ECS and Modal fit better, and managed harnesses increasingly take the problem off your hands. Nobody’s competitive advantage is patching a fleet of agent workers.

The **sandbox** is isolated compute for untrusted, model-generated code. Agents are now the largest source of untrusted code in most products that ship them, and a default Docker container is not a sandbox for adversarial code.



SANDBOX	ISOLATION	NOTES (LATE 2026)
E2B	Firecracker microVM	Most mature SDK. The default for ephemeral code execution.
Modal	gVisor	The realistic choice if the agent needs GPUs inside the sandbox.
Daytona	Persistent workspaces	Fast startup. Production codebase went closed source in June 2026.
Vercel, Cloudflare	Platform sandboxes	Attractive if you’re already there. Vercel’s went GA in January 2026.

CREDENTIALS & THE BLAST RADIUS

At Orchestra we coupled the harness to the sandbox, so every session runs in its own dedicated sandbox. OpenAI argues the opposite in *The next evolution of the Agents SDK* (April 2026), for three reasons:



SECURITY

Assume prompt injection. Credentials never sit where model-generated code runs.



DURABILITY

State lives outside the sandbox and is snapshotted. A dead container resumes from the last checkpoint.



SCALE

One run, one sandbox or many. Subagents get their own isolated environments.

Either way, the detail that separates good setups from bad is **credentials**. The agent needs authenticated APIs, but secrets inside the sandbox can be read and exfiltrated. Inject credentials at the network boundary and lock egress to an allowlist.

SANDBOX_POLICY.YAML

KEYS AT THE BOUNDARY, NOT IN THE BOX

```
sandbox:
  image: python:3.12-slim
  egress:
    default: deny
    allow:
      - host: api.github.com
        inject_header: { Authorization: "Bearer ${secrets.github_token}" }
      - host: acme.snowflakecomputing.com
        inject_header: { Authorization: "Snowflake Token=${secrets.sf_oauth}" }
  env: {} # nothing secret is ever mounted into the sandbox
  credentials:
    ttl_minutes: 15 # short-lived, scoped per session
```

For data people: this is why running agents inside a production Airflow cluster is a disaster. That cluster holds the keys to your castle.

The harness can run **inside** the sandbox (Claude Code in a container) or **outside**, treating the sandbox as a tool. Outside is easier to secure, but both are similar attack surfaces. The difference is that the sandbox runs arbitrary scripts.



HOW ORCHESTRA HELPS

Every Orchestra Agent Runtime session gets a **dedicated sandbox**, separate from your data workloads. Secrets stay in the **control plane** and reach agents as short-lived, scoped credentials.

OBSERVABILITY & EVALS

Agents are non-deterministic, long-running and capable of spending real money. A fun combination to run in production without visibility.

Agent observability means tracing every session end to end: every model call and tool call, inputs and outputs, latency, tokens and cost. The main tools are **Langfuse**, **LangSmith**, **Braintrust** and **Arize Phoenix**, with Datadog and other incumbents adding LLM features. OpenTelemetry's GenAI conventions are slowly making traces portable.

The telling signal: **ClickHouse acquired Langfuse in January 2026**, arguing that LLM observability is fundamentally a data problem. Traces are high-volume event data, evaluating quality is analytics, and finding the agent burning budget is a cost-attribution query.

THERE'S ALWAYS A DAG. YOU JUST NEVER SEE IT IN THE LOGS.

Trace a session and you see the DAG of what the agent did inside the loop. That's why it's called *agent orchestration*.

Evals are the other half: a regression suite for behaviour, rerun whenever the prompt, model or tools change. Without them every model upgrade is a vibes-based deployment.

HOOKS.PY

COST PER SESSION, ENFORCED AS IT RUNS

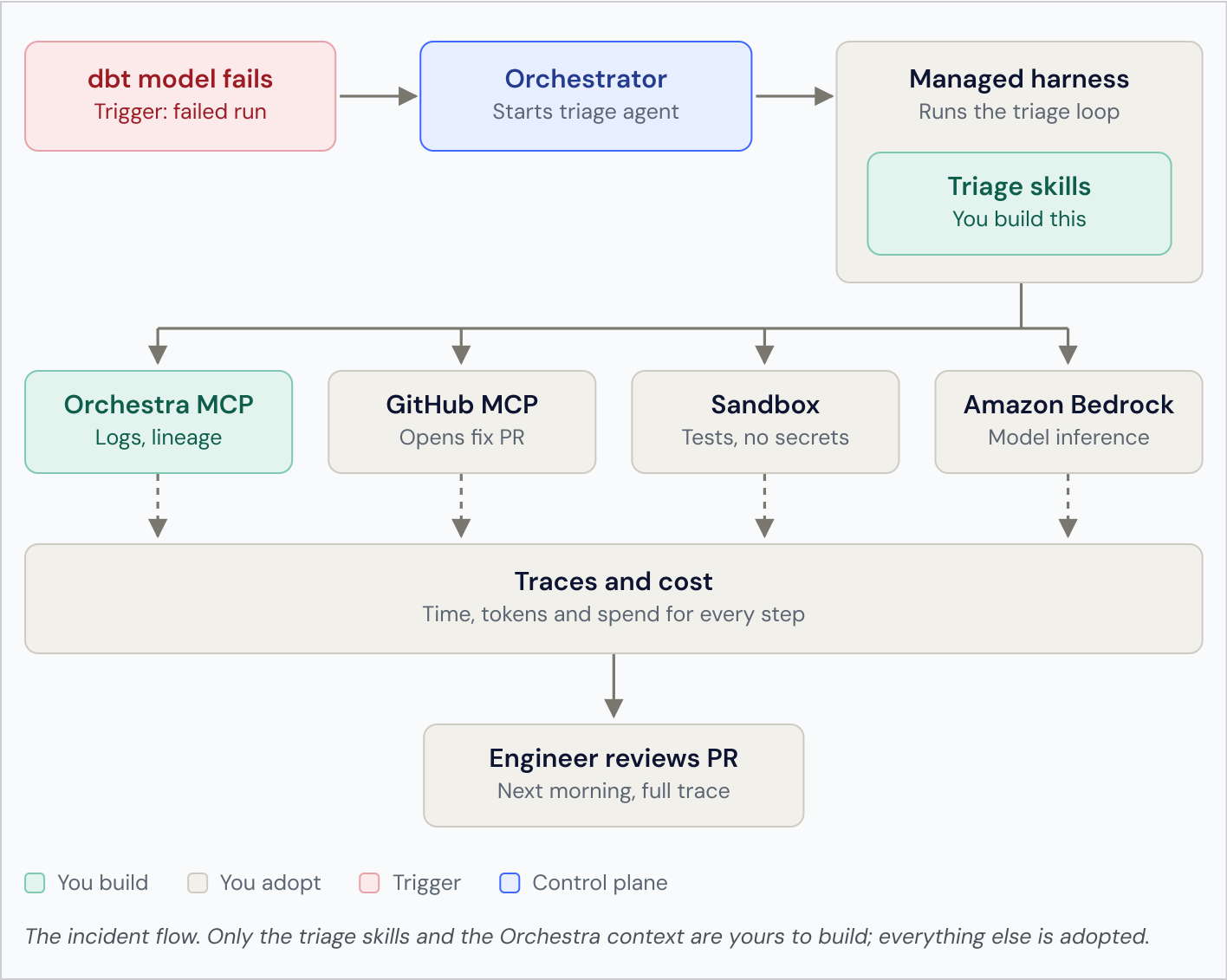
```
@harness.hook("after_model_call")
def track_cost(session, call):
    session.trace.log(model=call.model, tokens=call.usage, cost=call.cost_usd)
    if session.cost_usd > session.budget_usd:
        session.pause(reason="budget exceeded")    # ask a human, don't burn money

@harness.hook("on_complete")
def attribute(session):
    metrics.emit("agent.cost_per_outcome", session.cost_usd,
                tags={"agent": session.agent, "outcome": session.outcome})
```

Token spend behaves like consumption pricing, and consumption pricing punishes success: your most-used agent is your most expensive. Cost per session and per outcome should be visible as the agent runs, not discovered in a spreadsheet at month-end.

ONE INCIDENT, SEVEN LAYERS

A dbt model fails overnight. Here is how the layers combine to have a fix waiting for review in the morning.



- 1 Trigger & orchestration.** The orchestrator sees the failure and starts a triage agent with the run context.
- 2 Harness & tools.** A managed harness loads the triage skills and calls tools through MCP: Orchestra for logs and lineage, GitHub for the fix.
- 3 Inference.** The model is served through Bedrock, because that’s where the enterprise’s commitment sits.
- 4 Sandbox & observability.** Test code runs with no access to secrets. Every step is traced, so a human can see what happened, how long it took and what it cost.

BUILD THE HARNESS, OR ADOPT IT

Most software companies should not build a harness from scratch. Adopt one, put your effort into the skills, tools and permissions that encode your domain, and make your API good enough that agents enjoy using it.

The companies that will struggle treat “agentic” as a chat panel bolted onto the UI, with none of the layers underneath. The most underrated layer is orchestration. Agents multiply the things that need triggering, sequencing, monitoring and paying for, and something has to see across all of it: the pipelines, the warehouse and now the agents acting on them. **A control plane isn’t optional in that world.**

POSTSCRIPT: WHY IS PI SO POPULAR?

Pi (pi.dev) is an open-source, MIT-licensed terminal harness created by Mario Zechner and maintained since April 2026 by Armin Ronacher’s Earendil. Its argument: a coding agent needs only four tools (read, write, edit, bash) and a system prompt under about 1,000 tokens. Everything else is an extension. OpenClaw runs on Pi under the hood, and Pi 1.0 (1 October 2026) added MCP through a single sandboxed script tool, cutting prompt tokens by around 40%.

Pi proves the point of this paper: the loop is tiny, and the differentiation is in the skills, tools and context you give it.



WHERE DO AGENTS FIT IN YOUR DATA STACK?

Orchestra is a serverless control plane for data and AI workflows. Run your pipelines, trigger agents when things break, give them context through MCP, and track what they did and what it cost.

[GETORCHESTRA.IO](https://getorchestra.io) ↗