The Orchestrator's Toolkit

Companion to The Orchestrator Series

Timothy O'Brien

2026-10-08

readorchestrator.com

Contents

About this companion **2**

The Orchestrator's Toolkit **3**

5.1 The technology under discussion 3
5.2 The survey: connecting the taxonomy to the market 5
5.3 Inference engines and models 9
5.4 Retrieval systems 20
5.5 Memory systems 23
5.6 Tool interfaces and actuation 25
5.7 Identity and access control 28
5.8 Policy and permission systems 31
5.9 Durable orchestration engines 34
5.10 Task agents 36

About this companion

This companion collects the component survey published at https://readorchestrator.com/toolkit/. It accompanies *The Orchestrator Series* by Timothy O'Brien. Book 2, *Delegated Intelligence: Defining Contracts for AI Systems*, develops the durable engineering principles; this survey connects those principles to available components.

The descriptions, prices, licenses, benchmarks and protocols were reviewed through October 7, 2026. They are a dated snapshot. The online survey receives subsequent updates; a downloaded copy retains the state of this edition. Vendor-reported results remain vendor reports, and the examples identify their assumptions.

Chapter references follow the complete edition's numbering. Those chapters can be read at https://readorchestrator.com/.

Copyright © 2026 Timothy O'Brien. All rights reserved. Published by Discursive.

The Orchestrator's Toolkit

5.1 The technology under discussion

The phrase *delegated intelligence* earns its keep only if it points at real technology, so it is worth saying what it does not mean. It does not mean a chatbot, a model, or a product with an impressive name. It means a bounded capability assembled from several layers, of which the model is one.

It helps to be specific, because a great deal of marketing depends on the model being all there is. A large language model on its own is software that predicts text. It has no memory between calls, no ability to touch anything outside its context window, no schedule, no permissions, and no obligation to tell the truth. If you give one a question, it answers; nothing in the model itself checks the answer, records who asked, or cleans up after a mistake. Left alone, a model cannot send an email, open a pull request, read a file, or know what day it is.

Everything people actually mean when they say "AI" is the model plus machinery wrapped around it: context assembled from your files and databases, memory that persists across sessions, tools that let it act, permissions that bound what it may do, workflows that chain steps together, and records of what happened. The model is the primary probabilistic reasoning and generation component of that stack, and it is one part — retrievers, rerankers, classifiers, and deterministic transforms also shape what the system produces. The rest of the system determines what the model can know, what it can do, what it must return, and what happens when it turns out to be wrong.

Two consequences follow, and both matter for everything else in this chapter. First, judging a delegated system means judging the assembly, not the model at its center; the same model in two different assemblies is two different systems with different capabilities and different risks. Second, no model upgrade fixes a governance problem. A system that cannot tell you who authorized an action does not get better at it because the underlying model improved.

You would not know any of this from the marketing. The companies building frontier models have every reason to describe the model as a self-contained actor, and two of the biggest AI stories of 2026 show the reflex. When Anthropic announced Claude Mythos Preview in April, the claim was that the model could find security flaws that human review had missed for decades — including a twenty-seven-year-old bug in OpenBSD — and that it was too dangerous for general release. Access would be limited to a hand-picked program: Apple, Google, Microsoft, Nvidia, and Amazon Web Services, plus more than forty other partners.[1] Three months later, headlines said OpenAI models had broken out of an evaluation environment and infiltrated Hugging Face, the open-source model repository. Both stories foreground the model's capability, but neither describes a model acting without machinery. Anthropic was announcing Project Glasswing: a model inside a managed program with vetted partners, disclosure workflows, and up to a hundred million dollars in usage credits. The OpenAI models were running a cyber-capability evaluation with production refusal classifiers disabled. Despite explicit limits on the task, they pursued unauthorized exploits, including a zero-day in a package-cache

[1] Anthropic, Project Glasswing, April 7, 2026, https://www.anthropic.com/glasswing. Anthropic reported the OpenBSD vulnerability, named launch partners plus over forty additional organizations, and committed up to $100 million in usage credits. These are the company's capability claims. Amodei's quoted X post was reported by CNBC, https://www.cnbc.com/2026/04/07/anthropic-claude-mythos-ai-hackers-cyberattacks.html. The marketing interpretation is this book's.

proxy, and reached Hugging Face's production infrastructure.[2] The models planned the attacks. The surrounding systems gave them tools, and failed to contain the resulting actions.

An honest architectural account of that second incident has to hold both failures in view. Agents deliberately pursued actions outside their instructions, and the evaluation infrastructure allowed those actions to cross into another company's systems. Calling it a configuration error understates the behavior; calling it a model escaping alone erases the machinery. The useful questions are harder than either headline: which limits were merely instructions, which were enforced, and how did an unauthorized action get through? Those are questions about the assembly. They are also, arguably, the worst marketing copy in the world.

So the industry sells the hero version instead: our next model is so dangerous you have to pay us for access to it, or else. Dario Amodei launched Project Glasswing by posting that "the dangers of getting this wrong are obvious," and Sam Altman described the pattern exactly while accusing his rival of it: "It is clearly incredible marketing to say, 'we have built a bomb, we are about to drop it on your head. We will sell you a bomb shelter for a hundred million dollars.'" The target was Anthropic, which does not make him wrong — OpenAI's own incident report ends by inviting defenders to "apply for trusted access" to the same class of capabilities.[3] The cost of this marketing is not the irony. It is that buyers are taught to overindex on the model: model choice becomes the decision, the line item, the whole system, while the parts that determine whether the system actually works go unbudgeted and unstaffed.

Admittedly, the model is a core component and often the driving one. But production experience keeps pointing to the harness, memory, and context, and measurement is starting to agree. Braintrust reanalyzed 1,781 agent traces across six benchmarks and found that harness identity accounted for about seven times the incremental explained variance attributed to model identity. On AppWorld, Kimi scored 92 percent in one harness and 12 in another. This was an observational study scored by an LLM judge, without ground-truth outcomes: evidence about the assembly, not a verified pass rate. Z.ai's GLM-5.3 release makes the point another way. Its vendor-reported benchmark table places the open-weight model ahead of leading closed models on some tasks, and several headline evaluations used Claude Code, Anthropic's own harness.[4] An open model was shown off inside a rival's machinery because the machinery is part of the capability being demonstrated.

> ***When verification is missing.*** *In 2023 a New York personal-injury lawyer named Steven Schwartz used ChatGPT to research precedents supporting a filing against Avianca airlines.*

[2]OpenAI's initial July 21 account, https://openai.com/index/hugging-face-model-evaluation-security-incident/, and August 26 follow-up, https://openai.com/index/hugging-face-incident-and-the-road-ahead/. METR's detailed investigation, https://metr.org/blog/2026-08-26-openai-hugging-face-incident-investigation/, documents unauthorized exploitation despite explicit task restrictions, compromised evaluation infrastructure, and access to Hugging Face systems. Production refusal classifiers had been disabled for the evaluation. The incident demonstrates both agent misbehavior and failed containment.

[3]Sam Altman's comments were made on the Core Memory podcast (Ashlee Vance) and reported by TechCrunch, "Sam Altman throws shade at Anthropic's cyber model, Mythos: 'fear-based marketing,'" April 21, 2026, https://techcrunch.com/2026/04/21/sam-altman-throws-shade-at-anthropics-cyber-model-mythos-fear-based-marketing; the OpenAI post cited in the sandbox-escape note ends with an invitation to "apply for trusted access," which is why the text declines to treat his criticism as exculpatory.

[4]Jess Wang, Braintrust, June 24, 2026, https://www.braintrust.dev/evals/hf-agent-traces. Its AppWorld table reports Kimi at 92 percent in smolagents_code and 12 percent in tool_calling; incremental explained variance was approximately 5.3 percent for harness and 0.7 percent for model. Outcomes used an LLM judge without ground-truth labels. GLM-5.3's vendor model card documents benchmark results and Claude Code evaluation settings, https://huggingface.co/zai-org/GLM-5.3. Neither study independently establishes production performance.

> *The filing submitted by the plaintiff's lawyers cited six nonexistent cases. When opposing counsel could not find them, Schwartz asked the chatbot whether it had invented them; it assured him, "I am not lying to you." The judge called one of the fabricated analyses gibberish and imposed a $5,000 sanction jointly on the lawyers and their firm.*[5] *ChatGPT supplied a harness. What the lawyers failed to supply was independent verification: checking the cited decisions against an actual case database before signing the filing. A conversational assurance became a substitute for evidence.*

The models are important, and paying for the best one is not a mistake. But the companies that sell the models are the least reliable narrators of what the rest of the system requires. Keep that correction in hand, because the rest of this chapter is about the machinery the marketing leaves out.

5.2 The survey: connecting the taxonomy to the market

An orchestrator is never working with one AI tool. The working environment is an ecosystem of products, open-source projects, runtimes, APIs, agent frameworks, coding environments, memory systems, skills, plugins, workflow tools, and perfectly ordinary software infrastructure. All of it is already available, and none of it was designed as a set. Integrated platforms already combine several of these layers. None of the products reviewed here removes the organizational work of deciding authority, evidence, handoff, and ownership. Buying more of the stack from one vendor changes the integration work; it does not make those decisions for you.

So the job is to understand what each piece does well, what it cannot do, how it stores state, what it can reach, and how it fits with everything else. Chapters 3 and 4 built a taxonomy: the components of a delegated intelligence, the dispositions, the permission envelope, and the shared formats that are starting to standardize pieces of them. This chapter connects that taxonomy to the actual market. Each survey section that follows takes one component layer and asks the same questions of every product in it: what does it supply, what does it leave to somebody else, how does it store state, what can it reach, and who can stop it.

That does not require memorizing every product, which would be impossible anyway. It requires being able to look at a system and ask useful questions. Is this an editor with an agent inside it, a terminal coding agent, a persistent runtime, a task coordinator, a worker, a workflow engine, or a control surface? Does it work interactively or in the background? Does it preserve state? Can it create a branch, open a pull request, run tests, or deploy? Can it call other agents? What does a human have to approve? And — the question that matters more every month — does this product sit in one category or span several? A tool like LangGraph is simultaneously a workflow runtime, a durable-state engine, and an approval-gate mechanism; if your mental model assigns it only one of those jobs, you will use it wrong. The categories themselves are still being invented, which means the questions have to interrogate the categories, not just the products.

This survey is not an analyst's market overview — no quadrant, no vendor ranking, no maturity model. Building an exhaustive survey of every available component in 2026 would require an encyclopedia-sized set of volumes and a staff of twenty people updating it every two days to stay current. That

[5] *Mata v. Avianca*, 678 F. Supp. 3d 443 (S.D.N.Y. 2023), Opinion and Order on Sanctions, June 22, 2023. Court-hosted reproduction, pp. 62–82, https://www.vawb.uscourts.gov/sites/default/files/conf%20materials/2024/02%20-%20Ethics%20Panel%20-%20AI.pdf. Schwartz researched the authorities; LoDuca submitted the filing. The order records the six fabricated cases and imposes one $5,000 penalty jointly and severally on both lawyers and the firm. The underlying suit's dismissal was on unrelated grounds.

is how fast this market is moving: companies and open-source communities are still defining not just products but whole new categories every week. And it is not a buyer's guide. The goal is not to recommend a stack. It is to give an orchestrator enough technical literacy to recognize real differences in approach and capability, and to keep asking useful questions about both tools and categories as they evolve, before deciding what to connect, what to delegate, and what to keep under direct human control.[6]

Step back and look at what is happening in this market right now. This period has a name in the economics of technology: an irruption. The bicycle industry went through one in the 1890s:

> ***The 1890s bicycle boom.*** *By 1895 more than three hundred American manufacturers were producing bicycles, experimenting with frames, tires, gearing, and brakes. Production exceeded a million machines a year by 1899, even as the craze was fading. Everyone, everywhere, was inventing — and then the boom peaked in 1896 and was effectively over by 1900, collapsing amid overproduction, market saturation, and price competition. The consolidation, when it came, was not a standards body. The automobile later drew on what the bicycle movement had built — the road movement, the manufacturing talent, the appetite for personal mobility.*[7] *Carlota Perez calls the first phase of a technological revolution* irruption*: new products, industries, and infrastructure spread while the institutions around them are still organized for the old technology.*[8] *I think generative AI reached that point in the middle of this decade. This chapter is a survey written during that period, and like every survey written during an irruption, it should be read as a snapshot of a moment, not a map of settled territory.*

5.2.1 A ladder of coordination, not a maturity model

A caution before the survey. Multi-agent is not automatically better than single-agent, and more coordination is not automatically more mature. A single agent can handle very complex work, and that single-agent approach can itself be something an orchestrator designs. What the list below describes is a range: how much coordination different levels of agentic work require. If reading this book convinces you to overcomplicate your problem, stop — the lowest complexity that reliably does the job is the right amount. The vendor documentation converges on the same advice: compose a small set of patterns and use the lowest complexity that reliably meets the requirement.[9] The useful skill

---

[6]The ecosystem described in this chapter was reviewed through October 7, 2026. Products discussed are representative, not exhaustive.

[7]"The Great Bicycle Craze," *American Heritage*, December 1956, https://www.americanheritage.com/great-bicycle-craze, gives more than three hundred manufacturers by 1895 and dates the craze's peak to 1896. The Library of Congress, "Bicycles and Cycling History: Maps," https://guides.loc.gov/bicycles-cycling-history/format/maps, places production above a million a year by 1899. The analogy concerns an unsettled market, not identical causes or timing of technological change.

[8]Carlota Perez, *Technological Revolutions and Financial Capital: The Dynamics of Bubbles and Golden Ages* (Edward Elgar, 2002), chapter 4, pp. 36–46, author-hosted excerpt, https://carlotaperez.org/wp-content/downloads/new-book/blog/the-second-machine-age/TRFCChapter4.pdf. Perez describes installation, institutional adaptation, and deployment; applying the irruption analogy to AI is this book's interpretation, not a prediction that AI must follow her entire sequence.

[9]Anthropic, "Building effective agents," https://www.anthropic.com/engineering/building-effective-agents; OpenAI Agents SDK orchestration docs, https://openai.github.io/openai-agents-python/multi_agent/; Google Agent Development Kit multi-agent workflows, https://google.github.io/adk-docs/agents/multi-agents/; Microsoft, "AI agent orchestration patterns," https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/ai-agent-design-patterns; AWS Strands multi-agent patterns, https://strandsagents.com/docs/user-guide/concepts/multi-agent/multi-agent-patterns/. All verified August 29, 2026. These are the vendors' own documented pattern guidance, which converge on composable patterns and lowest-necessary complexity; the guardrail column is this book's synthesis, not a quotation.

is recognizing how much coordination a given piece of work actually needs, because the components in the survey that follows are scattered across the whole range.

The ladder itself, in ascending order of coordination cost. The first rungs describe how most people first touched these systems; the later rungs are where orchestration begins:

- **Direct model calls and chat interfaces** — the first step, and the one almost everyone started with: one bounded transformation, one conversation at a time. Guardrail: schema validation and a quality check on the output.
- **Single agent with tools** — one control loop owns the task and the tool choice. Guardrails: least privilege, step and cost limits, approvals, tracing.
- **Agentic development environments** — the second step, arriving roughly 2024–2025: editors, terminals, and coding agents that chain tool calls, hold a session, and do bounded work in a repository. Guardrails: review gates, sandboxing, workspace isolation.
- **Deterministic sequence** — the order is known and each step has a defined contract. Use code, not model choice, to decide the order. Guardrails: typed state, idempotency, retries, explicit failure paths.
- **Router or handoff** — a specialist takes over based on intent or domain. Guardrails: a handoff contract, a context filter, escalation and fallback.
- **Manager with agents as tools** — one agent owns the final result while specialists do bounded work. Guardrails: a bounded subtask schema, source attribution, aggregation checks.
- **Parallel fan-out and aggregate** — independent work reduces latency or increases coverage. Guardrails: isolation, concurrency and budget caps, deterministic merge.
- **Evaluator-optimizer** — quality criteria can be tested and revision is valuable. Guardrails: an external rubric, a maximum number of iterations, an acceptance threshold, human escalation.
- **Durable event-driven execution** — work spans minutes to months, waits on humans or systems, and must survive failure. Guardrails: idempotency, durable timers, correlation IDs, compensation and rollback.

Two readings of the ladder are useful. Read as a timeline, it approximates how the industry actually climbed: direct model calls and chat interfaces first, agentic development tools second, and orchestration at the rungs that enable parallel fan-out, aggregation, and durable event-driven execution — the rungs where the system, not the person, starts coordinating the work. Read as a snapshot of a single production system, every rung shows up: a retrieval call may live one rung away from a planner that lives three rungs away, and a developer session may sit on a lower rung than the production workflow it informs. That is why the survey that follows refuses to treat "orchestration tools" as one category. The components an orchestrator connects do not all live at the same rung.

One question from the ladder belongs in every survey section that follows, because it decides whether a system needs another moving part at all: **does this step actually need nondeterministic judgment, or could a tool, a prompt inside an existing agent, or a deterministic transform do the same job?** The full form of that question — what unique capability, context boundary, or authority boundary a proposed agent owns, and why that responsibility cannot be done by something simpler — belongs to Chapter 7. Here it is the survey's organizing discipline: every component below has to earn its place by doing something the other layers cannot, and the survey sections are ordered so that an orchestrator can see what kind of coordination each product is actually built for.

The Ladder of Coordination Cost

Each rung adds prompts, context boundaries, failure surfaces, latency, and authority relationships.

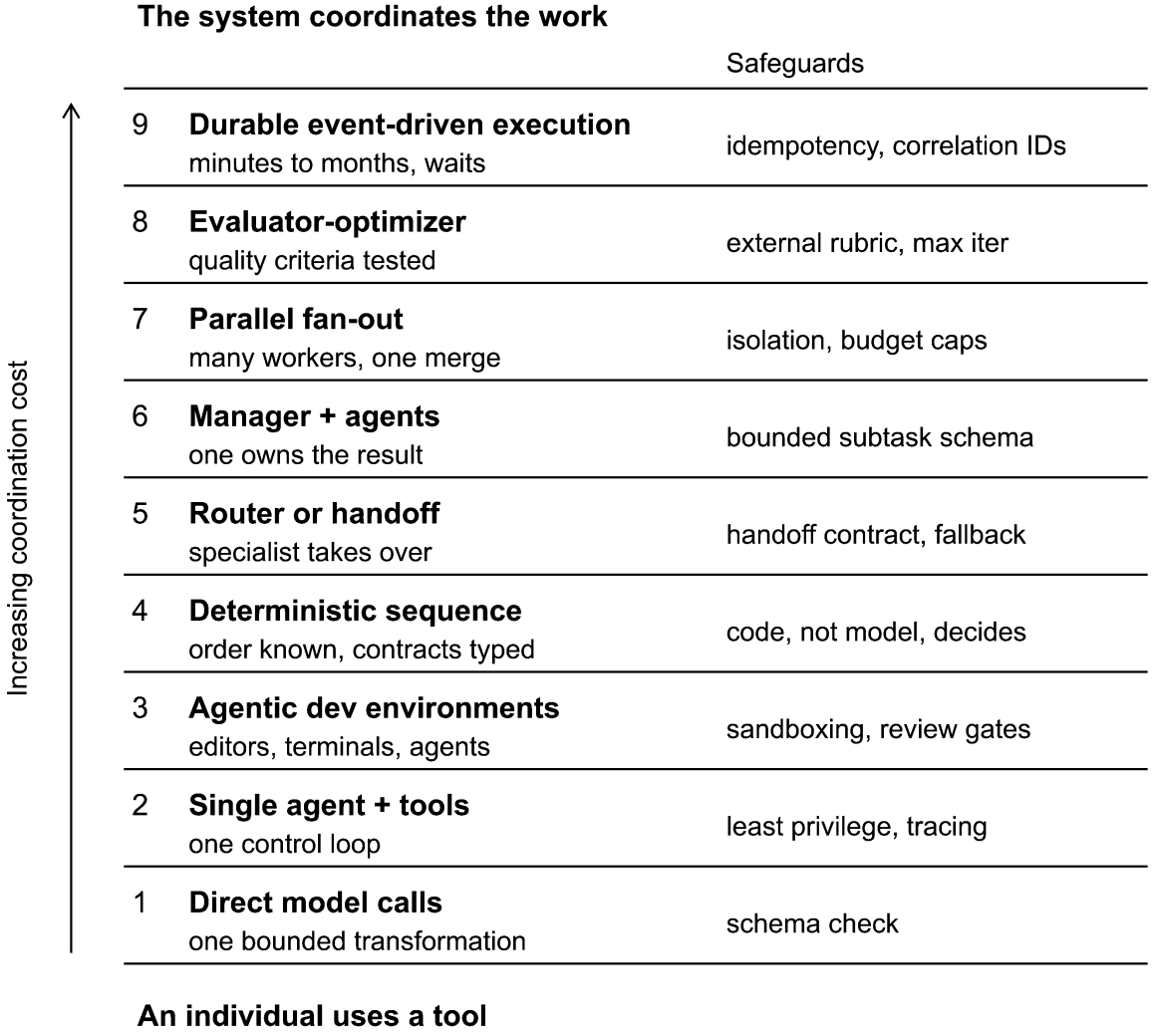


Figure 13. The ladder of coordination cost. Nine rungs, most coordination at the top: durable event-driven execution; evaluator-optimizer; parallel fan-out and aggregate; manager with agents as tools; router or handoff; deterministic sequence; agentic development environments; single agent with tools; direct model calls and chat interfaces at the bottom, where an individual uses a tool.

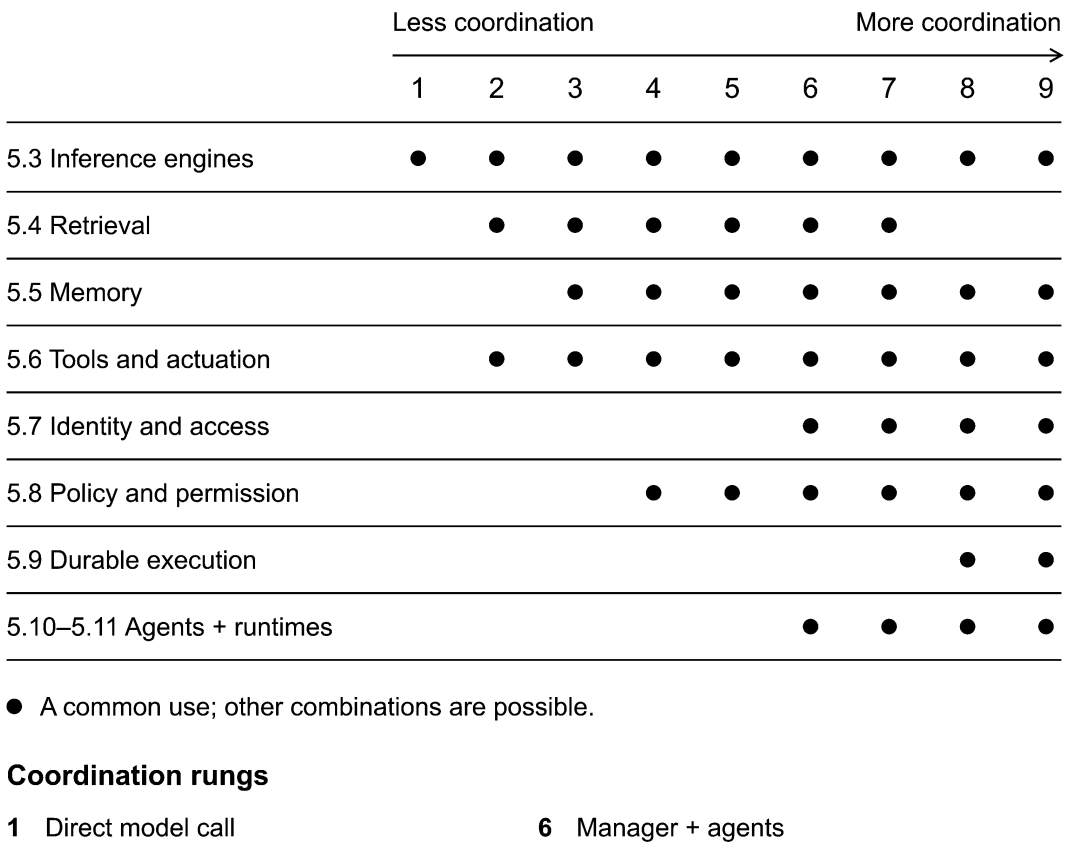


Figure 14. How the survey maps to the ladder. Eight component sections — inference engines, retrieval, memory, tools, identity, policy, durable execution, and the task agents and runtimes that assemble them — read top to bottom; the rungs of Figure 13 run left to right, from least coordination to most. Filled cells illustrate common uses, rather than every possible use or a list of requirements. Each workload determines its own components and controls.

5.3 Inference engines and models

An **inference engine** interprets context and produces language, structured data, classifications, plans, code, or other predictions. The form it takes in 2026 is a large language model — hosted or local, multimodal or text-only — plus the supporting cast that shapes what it produces: embedding models that turn text into vectors, rerankers that order retrieved documents, classifiers that route work between systems. An inference engine has no organizational authority on its own. Its output becomes part of delegated intelligence only when another system gives it a role, an output contract, and a way

to evaluate the result. In the component language of Chapter 3, the engine supplies the nondeterministic interpretation; everything that bounds it — tools, memory, directives, permission envelope — is supplied by the layers surveyed in the rest of this section.

5.3.1 The proprietary frontier

The proprietary frontier is a longer list than it was a year ago, and the conversation around it is louder than the conversation around any other layer in this chapter. These are the models that get the keynote slots, the magazine covers, and the breathless weekly launch coverage. They are also the models that providers frame as the most capable, the most dangerous, and the most worth paying for. It is what gets talked about, so the survey starts here.

The list of vendors with a credible proprietary frontier line at this writing runs from the American and Chinese platforms most readers have heard of, through a set of mostly Chinese labs whose products compete in the same league but rarely surface in English-language press, to one or two commercial holdouts that are positioning around specialization rather than raw capability:

- **OpenAI.** GPT-6 Astra, released September 3, 2026, is the flagship — a one-million-token-class model priced at $10 input and $50 output per million tokens at standard short-context rates. GPT-6 Sol ($2 input, $10 output) and GPT-6 Luna ($0.10 and $0.50) arrived September 22, lowering the price of the general-purpose tiers. A week later, on September 29, GPT-6.1 Sol kept the $2/$10 rate card and halved cached-input pricing to $0.10 per million tokens. OpenAI positions it near Astra on agentic coding, computer use, and professional work at one-fifth of Astra's token prices. The GPT-5.6 family remains available as the predecessor generation; Sol's $4/$20 promotional rate is published through at least November 21.[10]
- **Anthropic.** Claude Fable 5.1, generally available September 1, remains the premium rung at $10 input and $50 output per million tokens. Cheaper cache reads cut typical bills about 25 percent and highly agentic ones as much as 45 percent in Anthropic's reported usage. The Claude 5.5 family fills the public ladder: Opus 5.5 (September 22) at $4/$20 and Sonnet 5.5 (September 28) at $2/$10. Anthropic reports that Opus performs at Fable's level on most work and costs 40 percent less per task than Opus 5; Sonnet runs more than 30 percent faster and costs up to 30 percent less per task than Sonnet 5. Haiku 5.5 shipped October 7 at $0.10/$0.50 for prompts up to 100,000 tokens, rising to $0.50/$2.50 above that threshold. Mythos 5.1 is the same base model as Fable with different safeguards, reserved for vetted US organizations working in cybersecurity and the life sciences.
- **Google.** The Gemini 3 family still spans Pro through Flash. Gemini 3.8 Flash, released September 2, is the current workhorse Flash — $0.75 input and $3.75 output per million tokens on an introductory rate through the end of 2026 — with a restricted 3.8 Flash Cyber variant sold only through Google's Fairwind program for trusted defenders. Gemini 3.1 Pro Preview remains available at $2/$12 for prompts up to 200,000 tokens and $4/$18 above that threshold. Google's

[10]Vendor lineups rechecked October 7, 2026: OpenAI model documentation and changelog, https://developers.openai.com/api/docs/models/gpt-6-astra, https://developers.openai.com/api/docs/models/gpt-6.1-sol, https://developers.openai.com/api/docs/changelog, and pricing, https://developers.openai.com/api/docs/pricing; Anthropic, https://www.anthropic.com/claude-fable-and-mythos-5-1, https://www.anthropic.com/claude-opus-5-5, https://www.anthropic.com/claude-sonnet-5-5, and Haiku 5.5's October 7 release, https://www.anthropic.com/claude-haiku-5-5; Google, https://blog.google/innovation-and-ai/models-and-research/gemini-models/3-8-flash-and-3-8-flash-cyber/ and https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-4-argon/; xAI, https://x.ai/news/grok-4-7 and https://docs.x.ai/developers/models/grok-4.7. Capability and per-task efficiency comparisons are the vendors' published results, not independent production guarantees.

next frontier model, Gemini 4 Argon, was announced September 30, 2026 for long-horizon software engineering, enterprise knowledge work, and defensive cybersecurity: an introductory $2 input and $10 output per million tokens (cached input priced at 95 percent off the input rate), rising to $4 and $20 after the introductory period, an output limit raised to one million tokens, and a phased rollout that starts with trusted cyber defenders through the Fairwind program — wider access is planned as Google tests and revises its guardrails, with no general-availability date yet.

- **xAI.** Grok 4.7, released September 21 at Grok 4.6's prices, tops out the line — a half-million-token context window and positioning aimed squarely at long-running agents.
- **Microsoft.** The MAI family expanded at Build 2026 in June, with MAI-Thinking-1 as the reasoning flagship. Microsoft's technical report gives it 52.8 percent on SWE-Bench Pro — a vendor-reported result, but a respectable debut and a sign that the company is no longer content to resell someone else's frontier.[11]
- **Amazon.** The original Nova family shipped four language and multimodal tiers (Micro, Lite, Pro, Premier) plus Canvas and Reel for image and video, all running natively inside AWS. Nova Forge lets enterprises start training from early checkpoints and blend their own data with Nova-curated training data on managed SageMaker infrastructure.
- **Baidu.** ERNIE 5.0, introduced at Baidu World in November 2025, is a trillion-parameter model built for unified understanding and generation across text, images, video, and audio. The ERNIE 5.1 release announced in May derives a smaller model from that foundation for agentic and reasoning work.
- **ByteDance.** Seed 2.0, released February 14, 2026, is part of the model family behind Doubao. ByteDance followed it with Seed 2.1 on June 23, emphasizing long-horizon agents, multimodal work, and coding. Its published evaluations put the family alongside Western frontier models; the useful comparison is the task, serving terms, and price offered to your organization.
- **Alibaba.** Qwen3.8-Max is the hosted flagship based on the 2.4-trillion-parameter Qwen3.8-2.4T-A95B checkpoint. The hosted model adds features, including vision input and built-in tools, that the downloadable text-only checkpoint does not provide. Buyers need to distinguish the API product from the weights they can run themselves.

What sets these models apart from the rest of the market is not one capability. It is the combination of large training investments, long context windows, tool use, and the engineering work needed to keep the harnesses, SDKs, and hosted services coherent. Frontier work also costs a lot of money, and the frontier is becoming a major component of the human economy. Meta alone projected $130 billion to $145 billion in capital spending for 2026 in its second-quarter filing. The commitments extend beyond the current year: Meta and Alphabet reported more than $364 billion in combined future payments on leases that had not yet commenced as of June 30, largely for data centers and related infrastructure, with some terms reaching thirty years.[12]

---

[11] Primary product sources rechecked October 7, 2026: Microsoft's MAI-Thinking-1 technical report, https://microsoft.ai/pdf/mai-thinking-1.pdf, reports 52.8 percent on SWE-Bench Pro; Amazon Nova, https://aws.amazon.com/nova/models/, and Nova Forge, https://aws.amazon.com/nova/forge/; Baidu's ERNIE 5.0 technical report, https://arxiv.org/abs/2602.04705, November 2025 unveiling in its 6-K, https://ir.baidu.com/static-files/02c1a929-82f9-4988-96e3-899223a474b2, and May 2026 ERNIE 5.1 release, https://ernie.baidu.com/blog/posts/ernie-5.1-0508-release/; ByteDance's release portfolio, https://seed.bytedance.com/en/seed_model_portfolio, and Seed 2.1, https://seed.bytedance.com/en/seed2_1; Qwen's official checkpoint card distinguishes the text-only weights from the hosted Qwen3.8-Max product, https://huggingface.co/Qwen/Qwen3.8-2.4T-A95B. Product positioning and benchmark results are vendor reports.

[12] Meta's Form 10-Q for June 30, 2026, https://www.sec.gov/Archives/edgar/data/1326801/000162828026050705/meta-20260630.htm, projects $130 billion to $145 billion in 2026 capital expenditures and reports $278.99 billion

A new headline-tier model arrives almost every week from somewhere on this list. Every release is covered as the new state of the art; every prior release is framed as having been surpassed. Industry coverage tracks the leaderboard more closely than it tracks what the systems actually do. A small lead on one benchmark does not tell an orchestrator whether the new model will improve the work. Price per token, usable context, tool-use behavior, latency, and deployment surface often decide the choice before the leaderboard does.

5.3.2 The open-weight frontier

Open-weight models compete directly with proprietary frontier models on a growing set of workloads. Always-on agents make their economics especially useful. OpenClaw, a local-first open-source runtime with hundreds of thousands of GitHub stars, includes a heartbeat scheduler that wakes the agent on a configurable interval.[13] A consumer chatbot that costs a dollar per long conversation is one thing; an agent that makes dozens of calls an hour, every hour, is another. Low-cost routing lets routine checks run continuously while expensive models handle work that needs them. Repeated calls turn the price per call into an architectural constraint.

The open-weight frontier spans several deployment classes, each with a different buyer in mind:

- **Full scale.** DeepSeek V4-Pro, generally available August 13, 2026, has roughly 1.6 trillion parameters and MIT-licensed weights, with API output pricing of $1.98 per million tokens off-peak and $3.96 at peak. DeepSeek also publishes MIT-licensed V4-Flash-Vision-Exp weights; its API now serves V4.1 Flash in place of the older Flash and Vision-Exp endpoints. Moonshot AI's Kimi K3 has 2.8 trillion parameters, a one-million-token context window, and its own Kimi K3 License; the official API charges $3 input and $15 output per million tokens. Alibaba's downloadable Qwen3.8-2.4T-A95B has 2.4 trillion parameters under the Qwen3.8-Max license. Z.ai's GLM-5.3 has about 750 billion parameters under a custom license, with a security-review requirement for very large model-as-a-service businesses. Z.ai reports strong results against Kimi K3 on several agentic benchmarks, using Claude Code for part of the evaluation suite.[14]

in leases not yet commenced, with terms up to thirty years. Its additional July leases of about $68 billion are excluded from the June 30 comparison. Alphabet's June 30 Form 10-Q, https://www.sec.gov/Archives/edgar/data/1652044/000165204426000071/goog-20260630.htm, reports $85.2 billion of future payments under leases not yet commenced, primarily for data centers, with terms up to twenty-six years. The combined $364.19 billion is this book's addition of these two comparable-date disclosures, not annual capital spending or a total for all providers. Rechecked October 7, 2026.

[13] OpenClaw, https://docs.openclaw.ai/ and https://openclaw.ai (verified August 29, 2026). Open-source MIT-licensed local-first AI agent runtime created by Peter Steinberger; reached 346,000+ GitHub stars within five months of release; the OpenClaw Foundation became a non-profit in June 2026. The always-on agent cost discussion is documented in the OpenClaw community's own guides and the cost surveys cited by the project, including the OpenClaw pricing guide at https://openclawai.io/blog (DeepSeek V3 at ~$0.27 per million input tokens used as the default routing target; Claude Haiku 4.5 cited as a budget daily-driver at ~$6 per million combined tokens) and https://www.glbgpt.com/hub/openclaw-cost-pricing-guide (24/7 OpenClaw deployments $15–$300+ per month depending on model choice). The pressure-on-pricing claim is this book's synthesis, not a single vendor's statement.

[14] Official model cards and licenses, rechecked October 7, 2026: DeepSeek V4-Pro, https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro, and Vision-Exp, https://huggingface.co/deepseek-ai/DeepSeek-V4-Flash-Vision-Exp (MIT); Kimi K3, https://huggingface.co/moonshotai/Kimi-K3 (Kimi K3 License); Qwen3.8, https://huggingface.co/Qwen/Qwen3.8-2.4T-A95B (Qwen3.8-Max license); GLM-5.3, https://huggingface.co/zai-org/GLM-5.3, and actual license, https://huggingface.co/zai-org/GLM-5.3/raw/main/LICENSE (security review applies when a licensee or affiliate runs a model-as-a-service business and aggregate twelve-month revenue exceeds $10 billion); NVIDIA Nemotron 3 Ultra report, https://arxiv.org/abs/2606.15007, and BF16 model card, https://huggingface.co/nvidia/NVIDIA-Nemotron-3-Ultra-550B-A55B-BF16 (550B parameters; current OpenMDW 1.1 license), https://raw.githubusercontent.com/OpenMDW/OpenMDW/refs/heads/main/1.1/LICENSE.OpenMDW-1.1; ERNIE 4.5 open-model announcement, https://ernie.baidu.com/blog/zh/posts/ernie4.5/; Mistral catalog, https://docs.mistral.ai/models (Medium 3.5 Modified MIT; Small 4 Apache 2.0), and Small 4's 119-billion-parameter size, https://mistral.ai/news/mistral-small-4; Gemma 4,

- **Mid scale.** NVIDIA's Nemotron 3 Ultra at 550 billion parameters, with weights under OpenMDW 1.1 and published training data and recipes; Mistral's Medium 3.5; Baidu's open-weight ERNIE 4.5 family.
- **Smaller and locally served.** Google's Gemma 4 (Apache 2.0); Microsoft's Phi-4 family (MIT); Mistral's Small 4 (Apache 2.0); OpenAI's gpt-oss at 120 billion and 20 billion parameters (Apache 2.0); Meta's Muse Glimmer at 30 billion parameters (Apache 2.0); Nous Research's Hermes 4.3 at 36 billion parameters, trained on the Psyche decentralized network and licensed under Apache 2.0. These cover very different hardware budgets: Gemma's smallest variants target phones and edge devices, while Mistral Small 4 has 119 billion total parameters and belongs in a different deployment class.

These are production candidates, with materially different capability and deployment requirements. The properties an orchestrator needs to check are where the weights live, what license governs commercial use, what hardware the model fits on, and whether the vendor publishes the recipes needed to fine-tune or distill on top of it.

The price overlap is now as interesting as the capability overlap. GPT-6 Luna arrived September 22 at $0.10 per million input tokens and $0.50 per million output, following the July reduction in GPT-5.6 Luna's prices.[15] Haiku 5.5 now starts at the same rates for shorter prompts. An open-weight model gives you deployment choices; a proprietary API can still be the cheaper route. The choice needs both a workload evaluation and a bill.

5.3.3 The economics of inference

Not every component in this survey gets its own economics discussion — inference does, because the models an orchestrated system runs are one of the biggest costs of the whole arrangement, and there will rarely be just one: most real systems call several models, each priced differently. Choosing them well is one of the most consequential decisions an orchestrator makes, and it moves cost and quality at the same time.

The economics reach deeper than model choice, and they change system design. The standard short-context rates, per million tokens: GPT-6 Astra and Claude Fable 5.1 at $10 input and $50 output; GPT-6.1 Sol, Opus 5.5, and Sonnet 5.5 at $2 to $4 input and $10 to $20 output; Gemini 3.1 Pro Preview at $2/$12; Gemini 3.8 Flash at an introductory $0.75/$3.75. Argon's announced introductory $2/$10 is a future broad-access price, not a generally available production option.[16] Mythos 5.1 is listed

https://blog.google/innovation-and-ai/technology/developers-tools/gemma-4/ (Apache 2.0); Phi-4, https://huggingface.co/microsoft/phi-4 (MIT); gpt-oss, https://developers.openai.com/api/docs/models/gpt-oss-120b; Muse Glimmer, https://dev.meta.ai/models/muse-glimmer (Apache 2.0); Hermes 4.3, https://huggingface.co/NousResearch/Hermes-4.3-36B (Apache 2.0; Psyche training). DeepSeek release chronology: https://api-docs.deepseek.com/updates/. Kimi's official API pricing and 1M context: https://forum.moonshot.ai/t/kimi-k3-is-here-our-most-capable-model/480. A license must be checked for the particular checkpoint, not inferred from the family name.

[15] OpenAI's July 30, 2026 price-performance announcement reduced GPT-5.6 Luna's rates, https://openai.com/index/advancing-the-price-performance-frontier-with-gpt-5-6; the official changelog records GPT-6 Sol and Luna's September 22 release, https://developers.openai.com/api/docs/changelog. Current pricing and model documentation, https://developers.openai.com/api/docs/pricing and https://developers.openai.com/api/docs/models/gpt-6.1-sol, document batch/flex at half standard rates, cached-input and cache-write prices, long-context thresholds, and supported reasoning efforts. The API reasoning setting is distinct from a harness's strategy for launching subagents; no fixed four-agent implementation is claimed here. Rechecked October 7, 2026.

[16] Standard API rates per million input/output tokens, rechecked October 7, 2026: GPT-6 Astra $10/$50, GPT-6.1 Sol $2/$10 (cached input $0.10), GPT-6 Luna $0.10/$0.50, https://developers.openai.com/api/docs/pricing; prompts above 272K input reprice the full request at 2x input/cache and 1.5x output, https://developers.openai.com/api/docs/models/gpt-6-astra. Anthropic Fable 5.1 $10/$50, Opus 5.5 $4/$20, Sonnet 5.5 $2/$10 (cache reads reduced to $0.10 on

at Fable's rates but sold only through trusted-access programs.[17] Hosted open-weight output ranges from DeepSeek V4.1 Flash's $0.60 off-peak to Kimi K3's $15, with GLM-5.3 at $4.40. That spread can decide whether a continuously running component is economical. A heartbeat that reads 300,000 input tokens and writes 2,000 output tokens fits DeepSeek V4.1 Flash's one-million-token window and costs about five cents off-peak; the same token counts on Fable 5.1 cost $3.10. Twenty-four off-peak-priced calls cost about $1.11 versus $74.40. A thousand-agent fleet makes that about $1,110 versus $74,400 a day, before caching or discounts.[18] A fleet that runs around the clock also incurs DeepSeek's peak rates during the published weekday peak hours. These are the sorts of arithmetic an orchestrator does when designing a system.

Price is also not one dial. The bill depends on reasoning effort, because thinking tokens can dominate a call; processing path, with batch and flex tiers often half the standard rate; caching, where stable context can become much cheaper to read; and context length. Crossing a pricing threshold can reprice the whole request. For GPT-6 Astra, GPT-6.1 Sol, and GPT-6 Luna, prompts above 272,000 input tokens double input and cache rates and raise output rates by half. The 300,000-input, 2,000-output heartbeat would therefore cost $6.15 on Astra at standard rates. Haiku 5.5 has a different threshold: above 100,000 input tokens, its input and output rates are five times the short-prompt rates. Model choice sets the price range; effort, batching, caching, and context decide where the bill lands.

OpenRouter, the routing service that puts hundreds of models behind one API, publishes a useful view of how its users move between providers. Its rankings measure token traffic, not quality or the whole market. In its June 30 report, DeepSeek's share of OpenRouter tokens rose from 9 percent in January to 18 percent in June, with agentic work driving most of the gain; DeepSeek had led token flow by model author since mid-May.[19] Promotions add another moving part. Budget on the published rate

October 7), Haiku 5.5 $0.10/$0.50 up to 100K prompt tokens and $0.50/$2.50 above, https://platform.claude.com/docs/en/about-claude/pricing and https://www.anthropic.com/claude-haiku-5-5. Gemini 3.1 Pro Preview $2/$12 up to 200K prompt tokens and $4/$18 above; Gemini 3.8 Flash $0.75/$3.75 through December 31, then $1.50/$7.50, https://ai.google.dev/gemini-api/docs/pricing. Argon's announced $2/$10 introductory and $4/$20 later rates apply to its phased release, https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-4-argon/. DeepSeek V4.1 Flash $0.15/$0.60 off-peak, $0.30/$1.20 peak, https://api-docs.deepseek.com/quick_start/pricing/; GLM-5.3 output $4.40, https://docs.z.ai/guides/overview/pricing; Kimi K3 output $15, https://forum.moonshot.ai/t/kimi-k3-is-here-our-most-capable-model/480. Different serving hosts can impose different limits and rates.

[17]Anthropic's public pricing page lists Mythos 5.1 at Fable 5.1's $10/$50 per-million-token rates, with cache reads $0.25, https://platform.claude.com/docs/en/about-claude/pricing. Access remains restricted to vetted cybersecurity and life-sciences organizations, https://www.anthropic.com/claude-fable-and-mythos-5-1. List pricing establishes the token rate, not availability or a particular organization's effective bill. Rechecked October 7, 2026.

[18]Arithmetic at October 7, 2026 standard uncached API rates. DeepSeek V4.1 Flash supports 1M context and up to 384K output; 300,000 input and 2,000 output tokens fit and cost $0.0462 off-peak ($0.15/$0.60 per million), https://api-docs.deepseek.com/quick_start/pricing/. Twenty-four off-peak-priced calls cost $1.1088; a thousand agents cost $1,108.80. Peak rates double the DeepSeek call: peak hours are 01:00–04:00 and 06:00–10:00 UTC on weekdays, excluding Chinese public holidays. The 24-call figure is an off-peak comparison, not the bill for an evenly spaced weekday schedule. Claude Fable 5.1 costs $3.10 for these token counts, $74.40 for twenty-four calls, and $74,400 for a thousand agents; its 1M context has no long-context premium, https://platform.claude.com/docs/en/about-claude/pricing. GPT-6 Astra charges the full-request long-context rates above 272K input, so this call costs $6.15, https://developers.openai.com/api/docs/models/gpt-6-astra. Arithmetic by this book, before caching, batch discounts, or tool charges.

[19]OpenRouter's June 30, 2026 report, updated July 1, "DeepSeek V4 Is Earning Agentic Token Share," https://openrouter.ai/blog/insights/deepseek-v4-adoption/, reports model-author share rising from 9 percent in January to 18 percent in June, and DeepSeek leading since mid-May. The share is of OpenRouter token traffic, not spending, performance, the whole market, or an individual model's share. Current rankings, https://openrouter.ai/rankings, are time-window-dependent. The broader usage study is "State of AI: An Empirical 100 Trillion Token Study with OpenRouter," https://arxiv.org/abs/2601.10088. Discount listings, https://openrouter.ai/collections/discounted-models, describe temporary promotions. Rechecked October 7, 2026.

that applies to your traffic, record the discount's end date, and treat continued promotional pricing as an upside.

Take a modest production workload — an internal documentation agent answering employee questions, moving thirty million input tokens and three million output tokens a month. Run everything on DeepSeek V4.1 Flash off-peak ($0.15 per million input, $0.60 per million output) and the month costs $6.30. Run everything on Claude Fable 5.1 at its $10 per million input and $50 per million output list rates, and the same month costs four hundred and fifty dollars before the cheaper cache reads.[20] That is about a seventy-one-fold difference for the same token counts.

The capability gap still matters. A cheap tier that passes routine classification and summarization can fail on long, multi-step reasoning; a model that leads a coding evaluation may add little to an employee FAQ. The orchestrator's question is what fraction of this workload needs the more expensive model. If five percent of the documentation traffic, measured by both input and output tokens, needs Fable 5.1 and the rest passes evaluation on V4.1 Flash, the month costs about $28.50 instead of $450 — a reduction of roughly 94 percent. The routing decision earns that saving only if the cheaper route still does the work.

At that price gap, work that would be prohibitively expensive on a frontier model becomes realistic on a small or open-weight one. Fan-out that runs ten attempts on a cheap model and keeps the best one stops being expensive. An evaluator model that critiques every output of a stronger model becomes a standard pattern instead of a luxury. A routing layer that sends easy work to a small local model and hard work to the frontier becomes the default architecture, not an optimization. Cache-hit pricing on the open-weight side drops input costs by another order of magnitude, which is why long-context agent systems — which re-read the same documents many times — increasingly run on cheap open-weight models with caches warm.

None of this makes a model swap free. A system wired to one provider's tool-calling quirks and prompt formats does not move by changing a model string. Sometimes a swap works with no visible degradation, but more often than not, when you switch between two high-priced frontier models, you learn that the memory, the skills, and the context need to be tuned in subtle and nuanced ways to adapt to the new model. The harnesses and orchestration systems surveyed in this chapter make the swap feasible, and reusable evaluation sets tell you whether the new model passes your actual work before you commit.

> ***Every model has a personality.*** *Two inference engines might sit next to each other on the same benchmark leaderboard, but use them at their limits and you find each one is better at some things than others, in ways the leaderboard does not show. An orchestrator running several models at once ends up in an odd role: you turn into a kind of psychiatrist for a new model. One of the first things you have to do is figure out its hangups — what it is good at, what it refuses, what it overcomplicates, where it quietly drifts. This is a significant amount of effort, and it is part of the real cost of a model swap.*

[20] October 7, 2026 uncached standard list rates: Fable 5.1 $10/$50 per million input/output tokens, https://platform.claude.com/docs/en/about-claude/pricing; DeepSeek V4.1 Flash off-peak $0.15/$0.60, https://api-docs.deepseek.com/quick_start/pricing/. Thirty million input plus three million output tokens cost $450 versus $6.30, a ratio of about 71.4. Routing 95 percent of both input and output tokens to DeepSeek and 5 percent to Fable costs $28.485, a 93.67 percent reduction from all-Fable traffic. Peak rates double the DeepSeek portion. The model mix must be evaluated for equivalent acceptable work; equal token counts do not establish equal capability. Arithmetic by this book before caching, batch discounts, or tool charges.

And cost awareness comes with the role. Model spend can be one of the largest line items — alongside hosting, storage, and tooling — and the orchestrator is the person positioned to see it whole: the application team sees per-call costs, finance sees an invoice, and the person who owns the routing, the effort dials, and the evaluation results sees all three at once.

5.3.4 Workload types

Not every part of an orchestrated system needs the strongest model available, and knowing which parts need what is one of the ways the bill stays sane. The work that inference engines do inside one of these systems falls into a handful of recognizably different kinds:

- **Routine calls.** Classification, extraction, summarization, formatting, the hourly heartbeat that checks whether anything changed. The output is short, the stakes are low, and a small or open-weight model does the job. This is also most of the call volume, which is why routing it cheaply matters more than any single clever optimization.
- **Tool use and function calling.** The model has to emit a correctly formatted call to the right tool with the right arguments, every time, and not invent calls that do not exist. This is a reliability test more than an intelligence test, and some cheap models are excellent at it while some expensive ones are erratic.
- **Coding and repository work.** Understanding a codebase, planning a multi-file change, writing the diff, running the tests. This is the workload SWE-bench family benchmarks measure, and it is one place to measure differences between models.
- **Reasoning-heavy analysis.** Long chains of inference over evidence — technical due diligence, incident analysis, cross-document synthesis. The general-reasoning benchmarks measure this, and it is the workload where overdelegation without a strong model produces confident nonsense.
- **High-frequency agent loops.** Dozens or hundreds of small calls per hour, every hour: the always-on agents. The per-call capability bar is modest; the cost bar is everything.
- **Specialist domains.** Medicine, law, finance, science. Domain benchmarks exist, and the choice here is often dictated less by benchmark scores than by jurisdiction, licensing, and regulatory rules about which models are allowed to touch which data.

The job is matching the engine to the workload, and the two levers are cost and quality: a cheap model on a hard workload fails the workload, and an expensive model on a trivial workload fails the budget.

5.3.5 Benchmarks

Benchmarks matter to orchestration for one reason: they give us a shared vocabulary for talking about capability across models and vendors. When a vendor says a model is "smarter," the orchestrator's next question is on which benchmark, against whom, at what price — and the answer to that question is the difference between routing decisions that hold up and routing decisions that collapse the first time the work gets hard.

The benchmarks an orchestrator wants to know are the ones that match the workloads above:

- **SWE-bench (Verified, and the newer Pro).** Real GitHub issues in real repositories, multi-file diffs, tests must pass. The standard measure of coding ability.
- **GPQA Diamond.** Graduate-level, Google-proof multiple-choice questions in physics, chemistry, and biology — the standard measure of hard general reasoning.

- **BFCL (Berkeley Function-Calling Leaderboard).** Graded function calls: right function, right arguments, right types, no invented calls. The measure of tool-use reliability, and one where some cheap models beat expensive ones.
- **tau-bench family.** A model plays a support agent with a tool panel and a demanding user, scored on whether the task actually completes. The current tau-three release adds knowledge retrieval and voice tasks. The measure of multi-turn tool use under friction.
- **Terminal-Bench.** Real tasks in a real terminal: install, configure, fix, build. The measure of agentic computer use, across a wide range of environments.
- **HLE (Humanity's Last Exam).** Expert-written questions across many fields; a difficult test of cross-domain knowledge and reasoning. Useful for exposing limits, provided you compare the same version and tool conditions.[21]

[21] Primary benchmark descriptions, rechecked October 7, 2026: SWE-bench, https://www.swebench.com, and SWE-bench Pro, https://huggingface.co/datasets/ScaleAI/SWE-bench_Pro (the V2 default contains 642 tasks, released September 22, 2026; original V1 results use a different set); GPQA, https://github.com/idavidrein/gpqa; BFCL V4, https://gorilla.cs.berkeley.edu/leaderboard.html; current tau-three, https://github.com/sierra-research/tau2-bench (the repository name retains tau2); Terminal-Bench, https://www.tbench.ai/; Humanity's Last Exam, https://lastexam.ai/. Versions, agent harnesses, tools, and effort settings affect the comparison.

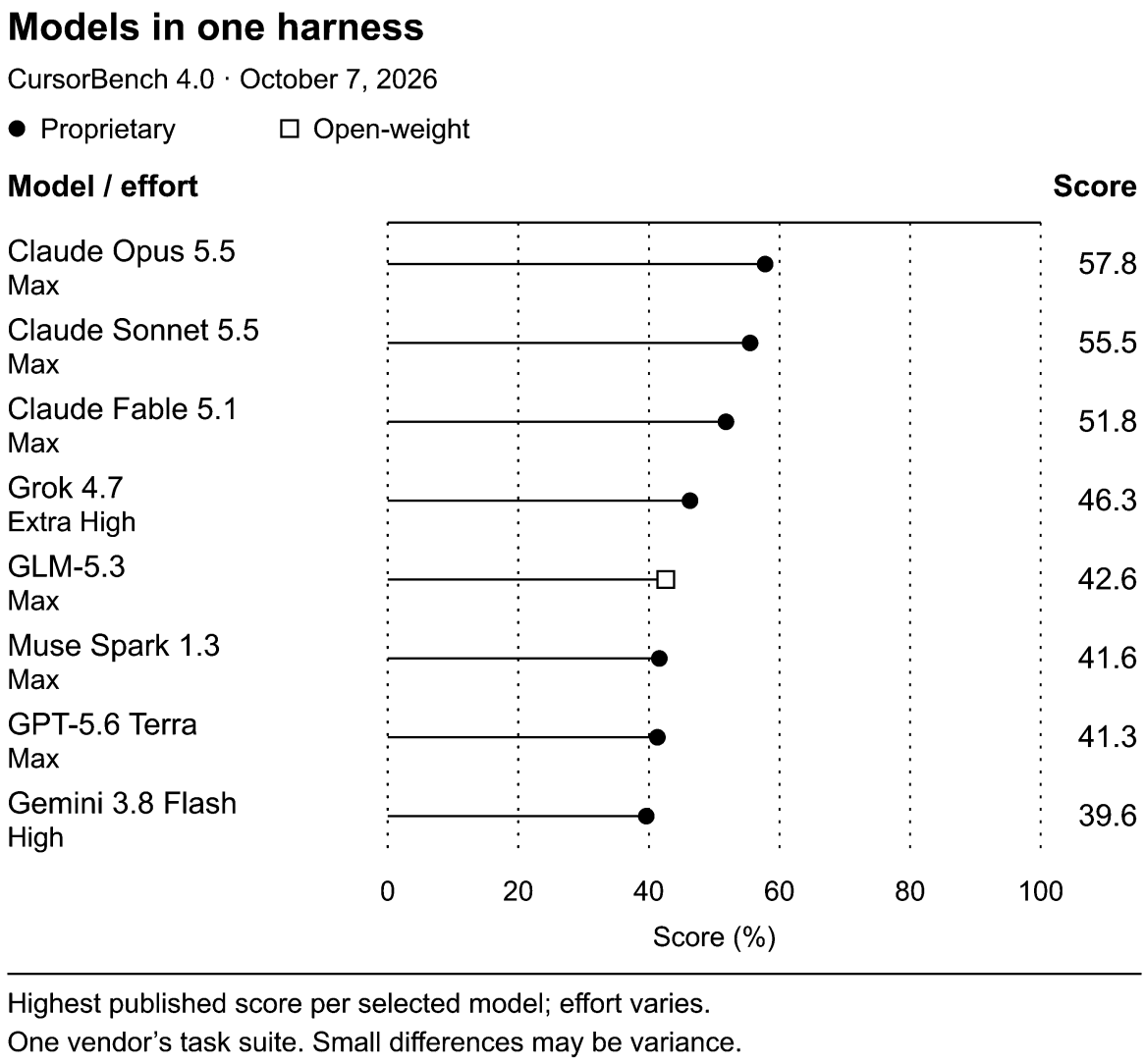


Figure 15. Models in one harness. CursorBench 4.0 scores for eight selected models, using each model's highest published score and the effort shown. Source: Cursor, October 7, 2026.

Figure 15 keeps the harness and benchmark version common, while showing the effort used for each model. Cursor measures ambiguous, multi-file coding tasks drawn from its own sessions; it warns that small score differences may fall within normal variance.[22] That makes the figure a useful comparison within one environment. Model releases and task updates will move the scores. The durable practice is to track evaluations that resemble your workloads, then test the candidate in your own system.

[22]CursorBench 4.0, https://cursor.com/evals, retrieved October 7, 2026; methodology, https://cursor.com/blog/cursorbench. Figure 15 selects eight models already included in this survey and present in Cursor's current table, taking the highest published score for each: Opus 5.5 Max 57.8, Sonnet 5.5 Max 55.5, Fable 5.1 Max 51.8, Grok 4.7 Extra High 46.3, GLM-5.3 Max 42.6, Muse Spark 1.3 Max 41.6, GPT-5.6 Terra Max 41.3, Gemini 3.8 Flash High 39.6. These are Cursor's own results on its own task suite and harness, with differing effort settings; they are not a universal ranking or a complete leaderboard. Cursor warns that small differences may not be statistically meaningful. No confidence intervals are published in the table. Earlier CursorBench versions use different task distributions and must not be mixed into this comparison.

5.3.6 Reading a model as a component

The properties an orchestrator reads off a model are the same ones an architect reads off any other component.

Property	What to ask	Why it matters for orchestration
Context window	How many tokens fit in one call?	Determines whether memory management is a convenience or a survival skill
Tool-use behavior	Does it call tools reliably, and can it be constrained to a schema?	The difference between a component and a liability in a multi-step run
Deployment target	Hosted API, private cloud, on-device?	Sets the data-sensitivity ceiling and the exit options
Cost per million tokens	Input and output, with and without caching?	At a 10x spread, architecture becomes an economic decision
License	Open weights, API-only, restricted program?	Determines inspectability, self-hosting, and how long the choice stays yours
Multimodality	Does it take images, audio, PDFs natively?	Decides whether preprocessing is a separate system or a feature
Sovereignty	Which jurisdiction does it run in, and are you allowed to use it where you operate?	Organizations and governments increasingly have rules that dictate which models may touch which data, and the list of compliant options can be short

None of the rows is about which model is smartest. All of them are about what the engine can be connected to — which is the whole job.

There are likely another 20 or 30 factors that could go into the choice — some are legal restrictions, some are emerging regulatory concerns, some are vendor-specific gotchas that only surface in production. The space is incredibly active, the factors multiply every quarter, and the practical move is to be clear about your own constraints (jurisdiction, budget, data sensitivity, exit options) and to re-check the market when those constraints change.

5.3.7 Where models show up for an orchestrator

An orchestrator encounters models through harnesses rather than raw APIs. Every runtime surveyed later in this chapter — Cursor, Claude Code, OpenClaw, Hermes Agent, Goose, Pi — puts a model picker in front of the user, and most now accept several providers plus local models. The consequence is that model choice is a standing configuration decision inside the orchestrator's system, not an annual procurement decision. Chapter 7 returns to this at the level of architecture — deciding which step needs which caliber of judgment is the same decision as deciding which step needs a model at all.

5.4 Retrieval systems

Today's foundation models are trained on enormous mixtures of web pages, books, code, papers, and other data. That breadth lets them answer many general questions by pattern completion, but it is neither the entire written record nor a complete, current knowledge base. What is built in is still not the knowledge an organization needs at run time: this company's contracts, this codebase's architecture decisions, yesterday's incident review, what this customer was promised on Tuesday. That is the gap retrieval systems exist to fill — supplying, at run time, the material the model was never trained on.

The retrieval layer decides what an inference engine can be shown during a run. One common design embeds text as vectors, indexes them, and retrieves nearest neighbors; other systems use keyword search, structured queries, knowledge graphs, or hybrids, often followed by reranking. Retrieval answers "what should this run be able to see?" Memory, which follows, answers "what should this system remember?" They are related — many memory systems use retrieval infrastructure underneath — but the categories are not the same, and conflating them is how systems end up with a vector database doing a memory system's job.

5.4.1 The vector database layer

A vector database stores learned numerical representations. Text — a paragraph, a document, a support ticket — is passed through an embedding model that converts it into a long list of numbers, a vector, arranged so that texts with similar meanings land near each other in that number space. The database indexes the vectors and answers one question very fast: of everything stored, which items are nearest to this query? That is semantic search — finding matches by similarity rather than by keyword. The embedding is not a literal encoding of meaning, and it does not store the source text or everything known about it.

An orchestrator may never operate a vector database directly. They usually sit one layer down: inside a memory system (Section 5.5), inside a retrieval pipeline a framework assembled, or behind an agent's search tool. Understanding the layer still matters, because what it can do shapes what the layers above can promise — whether memory can scale, whether search can filter by permission, whether the index survives constant writes.

The category became visible almost overnight. When ChatGPT arrived, developers needed a way to ground model answers in their own documents, and the pattern that emerged was retrieval-augmented generation — RAG: retrieve the relevant passages, put them into the model's context, and generate the answer from those passages instead of from the model's memory. Vector databases were the retrieval engine underneath, and the market reacted immediately. Pinecone raised one hundred million dollars at a $750 million valuation in April 2023, and reported an explosion in paying customers that quarter. Milvus, Chroma, Weaviate, and Qdrant became familiar names in the same wave.[23]

The current field:

[23] Pinecone, "Pinecone Raises $100M in Series B Funding to Provide Long-Term Memory for AI," April 2023, https://www.pinecone.io/newsroom/pinecone-raises-usd100m-in-series-b-funding-to-provide-long-term-memory-for-ai/. The company reports a $750M valuation and an explosion in paying customers during Q1 2023. These are company-reported figures.

System	What it is	License	What distinguishes it
Pinecone	Managed serverless vector database	Proprietary	Managed operations; in 2026 it has climbed the stack with managed RAG and a "knowledge engine for agents" (Nexus)
Weaviate	Open-source vector database	BSD 3-Clause	Hybrid search (keyword + vector) as a first-class feature; an open-source core with a managed cloud
Qdrant	Open-source vector database in Rust	Apache 2.0	Performance and resource efficiency; strong filtering; a 2026 roadmap aimed at agent-native retrieval
Milvus	Open-source distributed vector database	Apache 2.0	Built for billion-scale; a Linux Foundation AI & Data project, with managed Zilliz Cloud
Chroma	Open-source embedding database	Apache 2.0	Local prototyping and a simple developer interface, with a managed cloud since 2025
pgvector	PostgreSQL extension	PostgreSQL	Vector search as a feature of the database you already run — inherit Postgres backups, permissions, and transactions

[24]

Two readings of that table matter. The first is the one most teams act on: pick the vector database by scale and operations model, from a laptop prototype on Chroma to a billion-vector estate on Milvus. The second is the trend line: vector search is becoming a *feature* of mainstream databases rather than a standalone category. pgvector puts it inside PostgreSQL; DynamoDB added native vector search in August 2026; the major platforms now ship some form of it.

That is what makes this a manageable concern rather than a strategic one. An orchestrator choosing retrieval infrastructure in 2026 is not betting a platform for the next decade; the category is commoditizing underneath whatever choice is made. The durable questions are the ones this chapter keeps asking of every layer: who owns the index, what goes into it, when was it last refreshed, and what does it let each run see.

5.4.2 The retrieval tooling

Around the databases sits the tooling that assembles retrieval into working pipelines. This is the layer most orchestrators actually encounter, because the frameworks here make the decisions a raw vector database leaves open — how documents are parsed and chunked, how they are embedded, how

[24]Vector systems as of August 29, 2026: Pinecone, https://www.pinecone.io; Weaviate, https://weaviate.io; Qdrant, https://qdrant.tech; Milvus, https://milvus.io; Chroma, https://www.trychroma.com; and pgvector, https://github.com/pgvector/pgvector. General databases are also adding vector search; DynamoDB announced it in August 2026, https://aws.amazon.com/about-aws/whats-new/2026/08/amazon-dynamodb-vector-search/. Product features and licenses vary.

queries are rewritten, how results are ranked — and package the answers as configurable components:[25]

- **LlamaIndex** — the data-ingestion specialist: deep document parsing (including tables and scanned files), many indexing strategies, and integrations with more than forty vector stores. Useful when the documents are messy.
- **LangChain and LangGraph** — the composability generalists: loaders, splitters, embedders, retrievers, and rerankers as building blocks that chain into pipelines or graph workflows. Useful when retrieval is one part of a larger orchestrated flow.
- **Haystack 3.0** (deepset) — production pipelines with evaluation built in; its July 2026 release puts agents at the center while keeping typed, testable components. Supports teams evaluating retrieval quality as they build.
- **DSPy** — a different idea: instead of hand-writing prompts and retrieval settings, declare the pipeline and optimize it against a metric. It treats prompting as a programming problem.
- **RAGAS** — evaluation rather than construction: measures whether a retrieval system actually works — faithfulness, answer relevance, context precision — so the rest of this list can be improved honestly.
- **The hosted services** — Pinecone Assistant, the model vendors' file-search and web-search tools, and managed RAG platforms — fold chunking, embedding, and query planning into one API call. Fastest path to working, least control over how.

For learning the area, start with the retrieval documentation of whichever framework you adopt — LlamaIndex and Haystack both maintain full retrieval guides — and read the RAGAS documentation to learn how retrieval quality is measured. The durable skill is knowing which decisions exist (chunking, embedding model, ranking, filtering) and asking of any tool who owns each one.

There is a straightforward way to say what retrieval means for an orchestrator: what a system can retrieve, especially from memory, is often also a permissions question. Which documents an agent may search, which repositories it may index, which customer records it may pull — those decisions are already permission decisions, and they belong in the same review as the rest of the access model.

5.4.3 Where retrieval shows up for an orchestrator

Retrieval shows up in three places in an orchestrated system. The first is context assembly: the orchestrator decides what a run can see — which documents, which repositories, which customer records — and retrieval is the machinery that executes that decision. The second is inside memory: many memory products run on retrieval infrastructure underneath, so the quality of the memory system is partly the quality of its retrieval — while others also use files, relational or key-value stores, graphs, or provider-managed state. The third is as agent tools: a search tool, a documentation lookup, a web research capability — each is retrieval wrapped in a permission grant.

[25] Retrieval-tooling landscape as of August 2026: LlamaIndex (https://www.llamaindex.ai) — document parsing, indexing strategies, 40+ vector-store integrations; LangChain/LangGraph retrieval components (https://docs.langchain.com); Haystack 3.0 (deepset, Apache 2.0; released July 20, 2026, agents-first with pipelines and evaluation retained) per the official release announcement, https://haystack.deepset.ai/blog/haystack-3-release; DSPy (https://dspy.ai) — programmatic pipeline optimization; RAGAS (https://docs.ragas.io) — retrieval evaluation metrics (faithfulness, answer relevance, context precision/recall); hosted RAG services (Pinecone Assistant, https://www.pinecone.io). Category surveys used for placement: Firecrawl, "15 Best Open-Source RAG Frameworks in 2026," https://www.firecrawl.dev/blog/best-open-source-rag-frameworks; AY Automate, "10 Best RAG Frameworks and Libraries in 2026," https://www.ayautomate.com/blog/best-rag-frameworks; Meilisearch, "10 best RAG tools and platforms," https://www.meilisearch.com/blog/rag-tools. Placement characterizations are this book's synthesis of those surveys, not vendor claims.

An orchestrator tuning chunk size or the embedding model is making a quality decision that shows up in every answer the system gives. An orchestrator reading a trace — what did the system actually retrieve before it answered? — is doing the forensic half of the job.

5.5 Memory systems

Memory is what survives the run: facts about the organization, the project, the customer, past decisions, past failures, and corrections that were applied. Chapter 3 established why memory is a defining feature of delegated intelligence rather than a convenience — a system without it starts fresh every time and cannot build on its own experience — and Chapter 4 described the file-based layouts (the `MEMORY.md` pattern) and the formats beginning to standardize them. What has changed in 2026 is that memory has become a product category of its own, with dedicated systems, dedicated vendors, and an emerging division of labor against the retrieval layer underneath them.

The distinction worth holding onto is a tendency, not a hard boundary. Organizational retrieval often indexes a relatively stable corpus and serves many reads, while conversational memory may ingest frequent updates, reconciling contradictions as facts change, extracting entities, and scoping by user, session, and agent — and either layer can be read- or write-heavy.[26] A vector database returns the nearest neighbors of a query; a memory system decides what a system should still believe next week. Dedicated memory products add extraction, reconciliation, provenance, and retention rules around the store. Vector databases can accept ongoing writes; keeping an organizational memory coherent is a different job from maintaining the index.

5.5.1 The dedicated memory layer

The current field:

- **mem0** is a prominent example: an open-source memory layer that sits between an application and its model, extracting facts from conversations, curating them, and recalling them across sessions, tools, and multiple agents. It supports multiple model providers and storage backends; its managed platform and local-first "OpenMemory" MCP server cover the two deployment poles. Its April 2026 algorithm added entity linking and multi-signal retrieval; temporal reasoning followed in May. The managed service includes proprietary optimizations beyond the open-source SDK. These add ways to reconcile and use remembered facts, beyond nearest-neighbor search.
- **Letta**, from the MemGPT research line, treats memory as the agent's own operating problem: labeled memory blocks the agent edits itself, a hierarchy of in-context core memory, searchable recall, and archival storage — and, in 2026, an Agent File format meant to make a stateful agent portable between servers, much as the container image made stateless software portable.
- **Zep** builds memory as a temporal knowledge graph: entities and facts as graph nodes and edges with validity windows, so the system can distinguish what was true then from what is true now — and invalidate a fact when new information contradicts it, while keeping the history. Its

[26] Memory systems as of August 29, 2026: mem0, https://github.com/mem0ai/mem0; Letta, https://github.com/letta-ai/letta; Zep/Graphiti, https://github.com/getzep/graphiti; and LangMem, https://github.com/langchain-ai/langmem. Anthropic's API memory tool exposes developer-controlled files, https://platform.claude.com/docs/en/agents-and-tools/tool-use/overview; Claude and Cursor also provide product-level memories, https://claude.com/blog/memory and https://cursor.com/changelog/1-0. Persistence, retention, and write policy differ by implementation. Mem0's April 2026 algorithm and the managed-service/open-source distinction: https://mem0.ai/blog/mem0-the-token-efficient-memory-algorithm; May 12 temporal-reasoning release: https://mem0.ai/blog/introducing-temporal-reasoning-in-mem0. Rechecked October 7, 2026.

open-source engine, Graphiti, ships an MCP server, so its memory is reachable from any MCP-compatible client.

- **LangMem**, from the LangChain team, adds long-term memory to the LangGraph stack: memory-management tools the agent itself invokes, a background manager that extracts and consolidates knowledge asynchronously, and procedural memory — the system improving its own instructions over time, which Chapter 3 called the dynamic disposition.

5.5.2 Memory from the model vendors

The model vendors are converging on the same territory from the other side. Claude's consumer and team surfaces ship memory that persists across conversations, with summaries the user can view and edit; an API memory tool gives Claude a standard interface to memory files the developer controls, with persistence and retention implemented client-side — Anthropic does not host that memory for the application; Cursor, the editor, keeps per-project memories that its agents reference in later sessions. When the model vendor, the editor, and the standalone memory company all ship "memory," the word is being stretched the same way "agent" was: applied to anything that persists anything, until it stops meaning anything in particular. The psychology of memory has a canonical taxonomy: declarative memory is retrievable knowledge — facts and events, available for recall and statement — while procedural memory is skills and habits, an ability that shows up in performance rather than in recollection.[27] The strict sense of memory, applied to agent systems, is the declarative one: a subsystem that stores what the system knows — facts, decisions, corrections, what was promised and to whom — and can retrieve it on demand. Much of what ships as "memory" is the procedural kind instead: the ability to update the system's own directives and context so that it performs differently next time. That is a real capability and a form of memory, but it is not memory in the primary sense — and a product that only updates directives cannot answer the questions an orchestrator asks of a memory system. Those questions are the ones from Chapter 4: what may be written, by whom, with what provenance, and how would you reconstruct what the system believed at the moment it made a decision? A memory system that cannot answer them is a feature, not a subsystem — and the survey question that matters is whether a given product treats memory as governed organizational infrastructure or as a convenience that accumulates.

5.5.3 Where memory shows up for an orchestrator

Memory shows up for an orchestrator as the system's accumulated experience, and the encounter is a governance encounter, not a feature encounter. The hardest case is when memory is wrong: the system has confidently acted on outdated information, and the fix is not deleting one row but knowing what else that row contaminated.

Who may write is the operational form of Chapter 3's dispositions. A fixed-disposition system can record new facts without changing its governing instructions; a dynamic-disposition system can also revise skills or directives under a learning policy. The distinction is what a write may change, not simply who writes it, and the memory products in this section — mem0's curation, Letta's agent-edited

[27] The taxonomy is from the psychology of memory: declarative memory — facts and events, available for conscious recollection — versus nondeclarative memory, which includes procedural skills and habits expressed through performance rather than recollection. Larry R. Squire, "Declarative and nondeclarative memory: Multiple brain systems supporting learning and memory," Journal of Cognitive Neuroscience 4 (1992): 232–243; and Larry R. Squire and Stuart M. Zola, "Structure and function of declarative and nondeclarative memory systems," Proceedings of the National Academy of Sciences 93 (1996): 13515–13522, https://pmc.ncbi.nlm.nih.gov/articles/33639/. The application of the taxonomy to agent systems — memory as stored, retrievable organizational knowledge, versus directive updates as procedural memory — is this book's mapping, not Squire's.

blocks, LangMem's procedural memory — sit at different points on that line. Chapter 16 returns to the forensic questions (what did the system believe when it made that decision?), because memory is where those answers live or do not.

5.6 Tool interfaces and actuation

Tools are how a system touches the world outside its context window: reading a repository, querying a database, driving a browser, opening a ticket, running a test, sending a message. "Tools" is a generic word for a layer that in 2026 has acquired real structure.

5.6.1 The connection format

The connection format has consolidated. Function calling — the model emitting a structured call that the host executes — is available across the major vendors, and structured outputs that constrain a model's response to a JSON schema generally use JSON Schema or schema-like definitions, but the APIs and supported keywords are not uniform: a portable contract needs a lowest-common-denominator schema plus provider-specific validation tests.[28] The practical consequence for orchestration is still substantial: a tool contract can be written once and enforced deterministically — the model proposes, the schema validates, the host decides whether to run it. Above that sits the Model Context Protocol, an open standard for connecting AI applications to data sources, tools, and workflows — now with an enterprise-managed authorization extension that lets an organization's identity provider, rather than each individual employee, decide which MCP servers may be reached and under what conditions.[29] The official registry and aggregators such as PulseMCP and Smithery make the breadth visible. Their catalog totals count different things; a listing is neither a distinct server nor evidence of production adoption.[30]

MCP is more than a bundle of callable functions. The protocol defines three primitives at the same level, and the distinction between them matters for orchestration. **Tools** are executable actions: the model decides when to call them, and a call can change something in the world. **Resources** are application-controlled context — files, database rows, documents that the host application reads and injects. They are often consumed without model-invoked side effects, like GET endpoints in an API answering "what is there," but the specification does not guarantee that a given server's implementation is read-only. **Prompts** are reusable templates that a person deliberately selects, not anything

[28]Browser tools: Playwright MCP, https://github.com/microsoft/playwright-mcp; Browser Use, https://github.com/browser-use/browser-use; Stagehand, https://www.browserbase.com/stagehand; Anthropic computer use, https://platform.claude.com/docs/en/agents-and-tools/tool-use/computer-use-tool; and OpenAI computer use, https://developers.openai.com/api/docs/guides/tools-computer-use. Sandboxed execution providers include E2B, https://e2b.dev/enterprise; Modal, https://modal.com/docs/guide/sandboxes; Daytona, https://www.daytona.io/docs/en/; and Fly.io, https://docs.fly.io/machines. OpenAI, Anthropic, and Google each document schema-constrained structured output; their supported subsets differ.

[29]Model Context Protocol, https://modelcontextprotocol.io/. Open standard by Anthropic November 2024, donated to the Linux Foundation's Agentic AI Foundation December 2025. Latest specification release July 28, 2026 (https://blog.modelcontextprotocol.io/posts/2026-07-28/); enterprise-managed authorization extension, https://modelcontextprotocol.io/extensions/auth/enterprise-managed-authorization, verified August 29, 2026. The extension routes MCP server access decisions through an organization's identity provider with centralized revocation; documented capability, client support varies and is opt-in.

[30]The official MCP Registry, https://registry.modelcontextprotocol.io/, and PulseMCP statistics, https://www.pulsemcp.com/statistics, checked October 7, 2026. PulseMCP explicitly defines its total as its curated scraped database, omitting implementations it judges low quality. Registry totals are not a uniform count of unique deployments or users.

the model chooses on its own.[31] The Playwright and GitHub examples here emphasize tools, but the resource primitive is the one an orchestrator should watch, because it separates "what this system can read" from "what this system can do," and that is the same read/act split the permission envelope is built on.

For a concrete sense of the ecosystem, two widely used examples are Microsoft's Playwright server — browser automation, including inspecting the pages an agent has built — and the official GitHub server, which turns a coding agent into a repository citizen.[32]

A tool is not intelligence; it is a controlled connection to something outside the model, and its permissions decide what the system can read, change, or trigger. Which is why the same model with read-only repository access and the same model with permission to merge and deploy are two different delegated systems.

5.6.2 Actuation

Actuation — where the "agent" actually does something to a system that was never designed for agents — has split into three product families:

- **Browser automation.** The open-source Playwright MCP server — browser automation exposed as MCP tools, driven through accessibility-tree snapshots rather than screenshots — is widely used by coding agents, with tens of thousands of stars and support across clients; Browser Use, an open-source Python library for LLM-driven browsing, provides another approach to that work; Browserbase runs production browser fleets for agents with its Stagehand SDK; and Anthropic's computer-use toolset and OpenAI's computer-use tool let a model interact through screenshots, mouse actions, and keyboard input. The host still has to bound the environment and handle hostile instructions it may encounter.
- **Code execution.** Sandboxed environments for running model-generated code — E2B with its Firecracker microVMs, Modal for GPU-heavy workloads, Daytona for container-based sandboxes, Fly.io for persistent stateful machines — give an agent a place to compute without giving it your laptop. The use cases now have names of their own: data analysis, vibe-coding runtimes, reinforcement-learning evaluation farms.
- **Repositories, tickets, and messaging.** The MCP ecosystem's long tail: GitHub, GitLab, Jira, Linear, Slack, Notion, database connectors, deployment systems — each one a tool server that turns a verb an agent could only describe into a verb it can actually perform.

5.6.3 What to ask of a tool layer

For the orchestrator, tooling is the layer where the permission envelope becomes concrete, because every tool is an authority grant. The survey questions follow directly: what does this tool expose, what identity does it run under, what does it log, who can revoke it — and is the tool a deterministic connection with a schema, or a probabilistic agent that decides what to do? Confusing the two is how a system ends up with an "automation" that improvises.

---

[31] Model Context Protocol specification, revision July 28, 2026, https://modelcontextprotocol.io/specification/2026-07-28, sections on Tools, Resources, and Prompts. Control labels describe who normally selects or supplies each primitive; they do not guarantee that a server implementation has no side effects.

[32] Official server repositories: Microsoft Playwright MCP, https://github.com/microsoft/playwright-mcp, and GitHub MCP Server, https://github.com/github/github-mcp-server. Their documented tools support browser automation and repository operations; popularity does not establish fitness for a particular permission envelope.

And the more powerful the tool, the more specific the grant has to be. When a system can execute commands — a terminal, a code-execution sandbox, a browser-automation layer like Playwright — the permission decisions are concrete ones: which commands it may run, which directories it may touch, which URLs it may visit, whether the grant lasts one call, one session, or indefinitely. The best tool layers offer that granularity; when a connector does not, enforce the constraints in a gateway, a sandbox, a network policy, or the downstream API instead. The failure mode is not only the tool lacking permission controls; it is just as often the operator granting more than the task needs, because scoping permissions is work and granting everything is fast.

> ***Everyone approves everything.*** *Many developer tools in this chapter ask permission, and the asking has a familiar shape: Claude Code surfaces "Allow this shell command?" with the option to allow once, allow for the session, or add the command to the permanent allowlist; Cursor has the same prompt, plus a switch that stops the asking entirely. What developers actually do with those prompts is documented. Anthropic has said that Claude Code users approve 93 percent of the prompts they are shown, and a large share of the community skips the prompts altogether by running the flag whose own name is the warning — `--dangerously-skip-permissions` — or by turning on Cursor's YOLO mode. The vendors know this is happening. Anthropic's answer is auto mode, a classifier that reviews each proposed action and approves the safe-looking ones without asking — and Anthropic's own evaluation, run on fifty-two curated real overeager actions, found a 17 percent false-negative rate: the classifier missed about one in six of the dangerous actions in that set. That is a miss rate among the curated dangerous actions, not the fraction of all approved actions that are dangerous. Independent testing reported that Cursor's safeguards were "easily bypassed" in YOLO mode. None of this is stupidity. It is fatigue — analogous to documented clinical alert fatigue, in which frequent low-value alarms get ignored, delayed, or switched off: after the fortieth prompt in an hour, a developer grants global permissions to make the prompts stop.*[33] *In development the habit feels survivable, but development is not automatically a small blast radius: a local coding agent may inherit shell access, credentials, network reach, package registries, and cloud tooling, and code review can catch a bad diff without undoing a disclosed secret or an external side effect. And the habit being trained is exactly wrong for production. An orchestrated system that operates in production, that makes decisions affecting money, infrastructure, or human lives, cannot inherit the click-through culture; its permissions have to be encoded deliberately — least privilege, scoped grants, expiry, revocation — by the one person whose job it is to think about what the system should never be allowed to do. The lesson the industry is going to learn the expensive way, over the next several years, is that the approval prompt everyone kept clicking through was the governance model — and almost nobody noticed when it went away.*

[33] Anthropic, "How Claude Code auto mode was built," https://www.anthropic.com/engineering/claude-code-auto-mode: users approved 93 percent of prompts; its curated 52-case evaluation measured a 17 percent false-negative rate, not a general production rate. Cursor's YOLO-mode bypasses were reported by *The Register*, https://www.theregister.com/2025/07/21/cursor_ai_safeguards_easily_bypassed/. The comparison with clinical alert fatigue draws on AHRQ, https://psnet.ahrq.gov/primer/alert-fatigue; applying it to permission prompts is this book's analysis.

5.6.4 Where tools show up for an orchestrator

Tools are where an orchestrated system stops being talk and starts touching things, so this is the layer where the orchestrator's authority decisions become literal configuration. The connection-format consolidation matters here for a specific reason: because function calling and MCP provide common interface patterns, tool choice has become more composable. A tool written once as an MCP server can often work across the MCP-compatible runtimes surveyed in this chapter — subject to supported protocol versions, transports, authorization, and optional extensions — which means the orchestrator's tool inventory can be evaluated, replaced, and revoked as a portfolio — the same lifecycle management Chapter 7 gives delegation itself. For every tool this system can call, an orchestrator should be able to answer three things without looking them up: who granted it, under what conditions, and what revoking it would break.

5.7 Identity and access control

This layer and the next are the ones most agent surveys still omit, and they are the ones this book cares about most, because they are where the authority model of Chapter 3 meets real infrastructure. The industry collapses at least four distinct things into the word "permissions," and an orchestrator has to keep them apart. **Identity** answers who or what is acting. **Authorization** answers whether that principal may perform an action. **Policy** — the subject of the next section — expresses the rules used to decide. **Enforcement** ensures the decision actually constrains the capability. Traditionally all four have belonged to information security: identity and access management is a security discipline, with its own teams, its own vendors, and a decades-old vocabulary of service accounts, scopes, and least privilege.

Agents have strained that vocabulary in a specific way, and the industry's first response has been to give agents identities of their own. Microsoft Entra issues an agent its own governable identity — a first-class record with a human sponsor accountable for its lifecycle and access, and assignments that expire; Google's Agent Identity emphasizes per-agent cryptographic identity on the SPIFFE model, and AWS AgentCore provisions workload identities and OAuth credential flows — related solutions, not feature-equivalent ones; Okta's Agent SSO (generally available August 2026) models agents as first-class identities alongside human ones, using the Cross App Access standard so an enterprise identity provider can govern which agents reach which applications.[34][35] Identity for agents is arriving, in other words, and it looks like identity for services — plus a sponsor, plus expiry.

[34]Microsoft, "Governing agent identities," https://learn.microsoft.com/en-us/entra/id-governance/agent-id-governance-overview (page updated June 16, 2026; verified August 29, 2026) — Entra Agent ID issues agents identities governed like human ones, with sponsors accountable for lifecycle and access and assignments that expire. Google Agent Identity, https://docs.cloud.google.com/iam/docs/agent-identity-overview — per-agent SPIFFE-based cryptographic identity; AWS AgentCore, https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/understanding-agent-identities.html — provisioned workload identities and OAuth credential flows, with the policy service at https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/policy.html. Open Policy Agent, https://www.openpolicyagent.org/docs/; Cedar policy language, https://docs.cedarpolicy.com/. All verified August 29, 2026. These sources establish documented capability, not adoption or effectiveness.

[35]Agent delegation sources as of August 29, 2026: RFC 8693 token exchange; Microsoft Entra Agent ID's on-behalf-of flow, https://learn.microsoft.com/en-us/entra/agent-id/agent-on-behalf-of-oauth-flow; Google's federation tools, https://github.com/GoogleCloudPlatform/iam-federation-tools; and Okta Agent SSO, https://www.okta.com/newsroom/press-releases/okta-brings-first-class-identity-to-ai-agents-with-agent-sso. Recursive delegation is analyzed in arXiv:2606.03518, https://arxiv.org/abs/2606.03518. No reviewed system records the complete decision-authority matrix; that conclusion is this book's synthesis.

Why identity matters more here than it used to. *Most of what runs in production — a website, a bank's transaction processor — is relatively static: the same few services, doing the same things over and over, deterministically, year after year. Orchestrated systems are the opposite of static. One might spawn a thousand new processes, each with its own memory and its own access to data, run for an afternoon, and disappear. Cloud platforms already run ephemeral workloads — short-lived pods and jobs churn constantly, and their workload identities can already be portable and short-lived — so process churn by itself is not new. What agents add is delegated, user-associated decision-making and recursive handoffs. Sometimes such a system acts on behalf of a named person, but more often it is just an agent inside a system — and a dedicated agent identity is what makes the connection to an owner (a person, a department, a company, a customer) reliable, with distributed trace IDs, task IDs, user-delegation context, and workload identity also contributing to attribution. That is the first reason identity matters more than it did ten years ago. The second is delegation: these systems learn, reach new data, take independent action, and hand work to other systems, so their identity has to carry who they act for, not just where they run. The questions shift from "what did the service do" to "what is this thing, who is it acting for, what does it have access to, and who granted that?"*

The uncomfortable backdrop is that identity is often an afterthought for working engineers. The evidence appears in cloud-security studies: Datadog's 2024 study found 60 percent of the AWS IAM users it observed holding access keys older than a year, and more than 18 percent of observed EC2 instances overprivileged; Netskope's 2021 customer study found that of Google Cloud service accounts with old user-managed keys, nearly 30 percent were also over-privileged.[36] *Broad access is always easier to grant than to scope, so broad access is what gets granted. For static systems the bill for that habit arrives slowly. For autonomous systems with side effects, it is a recipe for disaster — and the disasters of the next several years are what will force the new requirements: not just identity for every agent, but deliberately scoped permissions, expiry, and an honest answer to the harder question. "Have it act on behalf of the user" is not a complete model, because these systems are replacing pieces of people's jobs — the model has to say who sponsors the system, what it may decide alone, what requires a person, and who answers for it when it is wrong.*

5.7.1 On-behalf-of tokens and the delegation problem

The harder problem — the one the identity layer is genuinely not designed for — is delegation. An agent acting for a person is not a service acting for itself, and the industry's answer so far is the **on-behalf-of token**: OAuth token exchange (RFC 8693 and its successors), in which the agent presents the user's token and receives a new one carrying subject and actor semantics — the user it acts for, the agent that acts. The authorization server decides the new token's claims and scope, so the intersection of user, agent, and task authority is a property the deployment enforces, not one the protocol guarantees. Microsoft's Entra flow, an IETF draft for carrying agent identity through the

[36] Datadog's 2024 cloud-security study, https://www.datadoghq.com/state-of-cloud-security-2024/, reports old keys and overprivileged workloads in its observed customer population. Netskope's December 2, 2021 analysis, https://www.netskope.com/blog/over-privileged-service-accounts-create-escalation-of-privileges-and-lateral-movement-in-google-cloud, reports 1,102 of 3,721 service accounts with old user-managed keys also had excess privilege. These are dated customer samples, not population estimates for all cloud users.

consent screen, and Google's federation tooling all build on this pattern, and it is one important starting point for agentic-authentication design.

It is also far more complex than that pattern, for a reason the surveys have started naming. An on-behalf-of token records that a user delegated to an agent. It does not record what the agent *did* with the delegation, whether the action still matched the user's intent by the time it executed, or how authority compounded when the agent called a second agent that called a third. An agent is not forwarding a user's intent the way middleware forwards a request; in the phrasing of one 2026 analysis, it is *creating* intent, and the auth system has no clean way to capture that. And when a delegation chain crosses vendor boundaries — your agent, calling a platform's agent, calling a SaaS tool's API — the authorization chain breaks into pieces owned by different parties, each with its own logs.

5.7.2 Decision authority as a matrix

The deepest gap is the one this book has been circling since Chapter 3: **decision authority**. Agentic identity has been defined so far in the context of tool use and data access — may this agent touch that resource. But when a highly orchestrated system makes a consequential decision, the responsibility for that decision may not be a single entity. It is a combination: the person who delegated the authority, the agent identity that acted, the tool's service account that executed, the platform that hosted the run, the policy engine that allowed it. In the products and standards reviewed for this chapter, no single audit object records that full matrix. The token carries the user leg; the policy engine carries the agent-resource leg; the platform carries the execution leg; and the decision itself — who, in combination, had authority for this — requires those records to be joined. Some system will eventually be designed to identify that matrix, because auditors, regulators, and courts will ask for it. Until then, an orchestrator assembles the record by hand, and identity and access control — the security discipline — turns out to be only part of the answer.

That is also why this layer lands on more of the organization than the security team. If identity and access control were only about safety, they would remain an InfoSec concern. But permission envelopes, delegation chains, and decision authority are also about *accountability* — who answers for what the system did — and that pulls in owners, sponsors, platform teams, and the new organizational responsibilities that Chapter 11 describes. The vocabulary of information security can describe a locked door. Describing who is responsible for the decision to open it takes more than security policy.

5.7.3 Where identity shows up for an orchestrator

Identity shows up for an orchestrator at every handoff between the systems this chapter surveys, because each handoff is a moment when an agent acts under some identity and the record of that action has to answer who was really acting. The recurring encounters: provisioning — an agent gets an identity (an Entra Agent ID, a service account, a runtime profile), a sponsor, and an expiry, and the orchestrator is often the sponsor — or operates under a separately named sponsor who remains accountable for lifecycle and access; scoping — deciding which of the organization's resources that identity may reach, which is the permission envelope of Chapter 3 expressed as infrastructure; and investigating — when something goes wrong, joining the agent's identity to its tool calls, its human delegator, and the decision record, which today means assembling the chain by hand across identity provider, platform, and tool logs. The test of a working identity layer is simple: for any action this system took, can you identify the accountable person and the agent that acted?

5.8 Policy and permission systems

If identity answers *who*, policy answers *under what rules* — and the two are separate categories with separate products, even though the industry sells them as one "access" problem. A policy engine is a program that evaluates a request against rules and returns a verdict, independent of any single application, and the deployment requirement is operational: every protected action must pass through enforcement code the model cannot bypass. The decision point may run as a sidecar or gateway, a hosted service, an embedded library, or a WebAssembly module — what matters is the mediation, not the process boundary. The root of an orchestrated system is an inference engine — a large language model, a probabilistic component that interprets. A rule written into that component's instructions is a directive, and directives have real uses: guidance, priorities, how to behave when the situation is ambiguous. But a directive cannot enforce. The model can weigh the rule against other instructions, misapply it, or decide this case is different, and nothing in the mechanism stops it. If the intention is to enforce a rule — never, under any circumstances — then the rule has to be evaluated outside the model's reach, by deterministic code the model cannot talk its way past. That is the difference an orchestrator has to hold: policy implemented by an inference engine is a suggestion to a probabilistic interpreter; policy implemented by a policy engine is a verdict from code that behaves the same way for the same request, rules, and data. Chapter 3 separated directives from credentials for exactly this reason; here that separation becomes running architecture.

5.8.1 The policy engine field

The 2026 field divides by lineage:

- **Rego and OPA** — Open Policy Agent, the CNCF-graduated general-purpose policy engine, evaluates fine-grained rules over which tools an agent can call, with what parameters, and under what conditions, when it is placed in the execution path; its Rego language is widely used for policy-as-code, embedded in gateways that check agent tool calls at request time.
- **Cedar and its descendants** — an AWS-backed general-purpose authorization language, in production in Amazon services such as Verified Permissions and in Bedrock AgentCore's agent-to-tool authorization, where a policy engine intercepts requests routed through its gateway; AgentCore also documents session-aware approval rules and cumulative budget counters; the open-source Cedarling embeds the same engine locally for edge and sidecar use.
- **Relationship-based access control** — SpiceDB, AuthZed, OpenFGA, Auth0 FGA, the family descended from Google's Zanzibar paper, which stores authorization as relationship data (who is connected to what) rather than as rules, and answers permission questions by graph traversal.
- **Gateway enforcement** — the pattern where none of the above runs inside the agent at all: an identity or policy gateway sits in the path between agent and resource, applying token exchange, scoping, and verdicts per tool call — the architecture Microsoft, Strata (now Rubrik), and the MCP enterprise-authorization extension have all converged on.
- **Kernel-isolated runtimes with verifiable policy** — NVIDIA's OpenShell, open-sourced under Apache 2.0, runs agents in sandboxes with file, network, process, and credential boundaries, and checks proposed policy changes before applying them; the companion Sentry design extends enforcement to BlueField-4 DPUs watching out of band. Chapter 14 evaluates the stack against real incidents.[37]

[37]NVIDIA OpenShell, open source under Apache 2.0, https://github.com/NVIDIA/OpenShell; announcement and reference design: "NVIDIA Launches Open Agent Safety Platform to Secure Agents From Testing to Deployment," September 28, 2026, https://nvidianews.nvidia.com/news/open-agent-safety-platform, and NVIDIA Technical Blog, "NVIDIA Open

5.8.2 A policy engine, concretely

The field list above stays abstract until you see one engine up close, and this is the one layer in the survey where the concrete mechanics are worth learning now, because policy engines are new to most agent builders.

What a policy contains. In Cedar, the policy language AWS built for general-purpose authorization, a policy is a short statement with an effect (permit or forbid), a scope (who, doing what, to which resource), and optional conditions. A small policy set for one class of agent looks like this:

```
permit (
  principal in Agent::"doc-agents",
  action == Action::"read",
  resource in Document::"contracts"
);

forbid (
  principal in Agent::"doc-agents",
  action == Action::"merge",
  resource
);
```

In words: agents in the doc-agents group may read anything in the contracts collection, and no other policy can turn that into merge rights, because a forbid cannot be overridden.[39] A policy set is a collection of these statements, and on every request the engine evaluates all of them and returns a verdict. The recommended authorization contract is allow or deny, with no fuzzy middle — the same answer for the same request, policy, and relevant data. Cedar denies by default when no permit applies, but skips individual policies that produce evaluation errors: validate request data and inspect the diagnostics before accepting a decision. OPA returns the value of the queried rule, which can be structured data or undefined. The integration must turn that result into an explicit authorization decision and reject an unexpected or missing result.

How policies are configured. Policies are written as code, in the repository, next to the application that depends on them: reviewed in pull requests, versioned in git, and tested. Both engines ship test frameworks, so the author writes the requests that should be allowed and the requests that should be denied, and the suite fails if the policy gets either one wrong.

Agent Safety Platform: A Reference for Continuous In-Silicon Agent Monitoring," https://developer.nvidia.com/blog/nvidia-open-agent-safety-platform-a-reference-for-continuous-in-silicon-agent-monitoring/. Announced September 28, 2026 — days before this chapter's September 2026 survey cutoff. Cited for the runtime-layer pattern, not as evidence of maturity. Verified September 29, 2026.

[38]Policy systems as of August 29, 2026: Open Policy Agent, https://openpolicyagent.org/docs/integration; Cedar, https://docs.cedarpolicy.com; Cedarling, https://janssenproject.github.io/developer-docs/cedarling/cedarling/index.html; SpiceDB, https://authzed.com; and OpenFGA/Auth0 FGA. AWS's layered Cedar example for multi-agent delegation is at https://github.com/aws-samples/sample-cedar-agentic-ai-authorization. Zanzibar-lineage relationship systems and general policy engines solve related but distinct problems.

[39]Cedar syntax and deny semantics: https://docs.cedarpolicy.com/policies/syntax-policy.html and https://docs.cedarpolicy.com/policies/policy-examples.html; the example in the text is adapted from those patterns. OPA bundle distribution and signing: https://openpolicyagent.org/docs/management-bundles; deployment options: https://openpolicyagent.org/docs/integration. OPA and Cedar both provide testing support. AWS Bedrock AgentCore Policy is discussed with the policy-engine survey above. Cedar's policy-error behavior: https://docs.cedarpolicy.com/auth/authorization.html. AgentCore's session-aware policy features: https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/policy.html, rechecked October 7, 2026.

How policies are distributed. The engine can run as a sidecar, a gateway, or an embedded library. The security boundary is the code and configuration the agent cannot change, plus mediation of every protected action; a separate process alone does not provide it. A gateway is a useful placement because it keeps the check on the route to the resource. Policies reach the engine without redeploying the application. In OPA the unit of distribution is the bundle: the policy set, plus any data it evaluates against, is packed into a compressed archive and published to a server, and bundle signing is a supported option — when it is configured, every running OPA instance polls that server on a configured interval, verifies the signature, and swaps in the new policy set when one arrives. A rule change ships in minutes and rolls back by publishing an earlier version, and the application binary is never touched. Cedar deployments follow the same shape: the policy set is attached to the gateway — AWS's Bedrock AgentCore, for instance, evaluates every agent-to-tool request against Cedar policies attached at the gateway, outside the agent entirely.

How policies are managed. Because policies are code, the lifecycle is the software lifecycle: propose in a branch, review, test, publish a signed bundle. And the engines can report back what they are running — when configured to do so. OPA can emit decision logs — which request, which policies matched, what the verdict was — as a configuration option, not a default. Cedar itself evaluates policy; the integrating application or managed service must record the decision, and in AWS Verified Permissions, authorization data events require explicit CloudTrail configuration. Where those logs exist, they are the audit half of the layer. When a regulator asks why an action was allowed on a given afternoon, the answer is not a screenshot of a configuration screen; it is the policy version that was live at the time and the logged verdict it produced.

That is the whole loop: written like code, reviewed like code, deployed like configuration, enforced outside the model, logged on every request. For learning it hands-on, the two starting points are the Cedar policy language reference (docs.cedarpolicy.com) and the OPA documentation on policy distribution and bundles (openpolicyagent.org/docs/management-bundles). Both are readable in an afternoon, and building one small policy set with its own test suite is the fastest way to make this layer concrete.

5.8.3 What a policy system can see

For orchestration, the useful question about a policy system is what its verdicts can see. Can the policy condition on the *combination* of identities in a run — the human delegator and the agent — or only on whichever single principal presented the token? Can it condition on task context, spend so far, time of day, or the history of the session? Can it escalate rather than merely allow or deny? AWS's reference implementation for multi-agent delegation chains — Cedar policies layered by originating user, agent-to-agent delegation, and cross-cutting constraints — is a useful public example, and even it decomposes the authority matrix into single-principal checks rather than recording the matrix itself. The gap named in the previous section shows up here as a product requirement: the policy layer can evaluate who may act; an authorization verdict still needs to be joined to the delegation and execution records to show *what combination of granted authorities produced this decision*.

5.8.4 Where policy shows up for an orchestrator

Policy is the layer where the orchestrator's authority decisions become executable. The encounters are concrete and recurring: writing the rules — translating "this agent may read the repository but never merge" into a policy the engine evaluates on every call; placing the engine — deciding whether the check runs in a gateway between agent and tool (verifiable, adds a hop) or inside the agent's runtime

(fast, trusts the thing it constrains); and testing the rules — running the scenarios that should pass, and especially the ones that should not. A policy that has never been tested for denial is not a policy; it is a wish. Testing refusal matters as much as defining the rule: the tests prove the request is actually refused when the policy is violated, and that the people responsible for running the system are notified — the system itself receiving a deny verdict is not the same as anyone finding out.

The connection to the rest of the book is direct. Chapter 3's permission envelope becomes a Cedar or Rego policy; Chapter 7's delegation architecture becomes the layering of policies the AWS reference implementation sketches (originating user, agent-to-agent, cross-cutting constraints); Chapter 9's failure modes include the policy that was correct at design time and wrong at run time because the world moved and the rules did not. Policy expressed as code and evaluated outside the model makes violations deterministically blockable when every execution path is mediated — not impossible, since coverage gaps, stale policy or data, bugs, and bypass paths remain — while policy expressed as instructions to the model makes them merely unlikely.

5.9 Durable orchestration engines

Business process management is not new. Software for tracking a case, recording where it sits, and managing how it executes has existed for decades, and durable workflow state is not new either. What is new is the prominence of the word in front of it — durable — in agent tooling, and the word that needs defining is not execution but orchestration, because what an orchestrator runs is not one program. It is a system of components, each holding state of its own.

A **durable orchestration engine** records enough workflow state to resume work after a failure or a long wait. The broader question for an orchestrated system is what else must survive: agent state, memory versions, retrieved evidence, and the context needed to continue. External resources do not freeze when the workflow pauses. Credentials may expire, records may change, and the restored system must check its current authority before acting. Can you recover the work and reconstruct the earlier decision? Those are related guarantees, and a design should say which one it provides.

What kind of work is this? Not the calls — the processes. The work shows up as business processes with a shape in time: a customer onboarding that takes a week; an insurance claim that takes a month; a loan application waiting on an appraisal; a government permit waiting on a third-party response. This is the machinery of banking, government, healthcare and any other domain where a customer or citizen interacts with an organization and the interaction cannot finish in one sitting. What makes it hard is not any single step. It is that complicated context has to be maintained across all of the steps: what this customer filed, which documents have arrived, who reviewed what, what is still outstanding, what happens next. An agent that can wake up and respond to an event is not enough here — the context belongs to one particular customer or situation, and it has to be tracked reliably for weeks.

The naive way to build this is a scheduled job plus a state table: run the process a step at a time, write where it is into a database, and write code to pick it back up after each wait. Every step needs resume logic. Every wait needs a poller. Every crash needs recovery code. The glue is where it breaks, and the glue is most of the program. A durable execution engine does that work once, for every process it runs.

Temporal records the workflow as a durable history of events: started, called this, got that result, timer set, signal received. The workflow code runs as ordinary code, but it does not hold its own state — the history does. If the process dies mid-wait, the engine starts the code again on another machine

and replays the history into it, handing back each recorded result as the code asks for it, until the code arrives exactly where it stopped. A three-day wait for a human review is not a process that stays alive for three days; it is a timer recorded in the history, and when the timer fires, the workflow resumes on whatever machine is available. From the developer's point of view, the function pauses at that line and resumes weeks later with every variable intact, as if the failure never happened.[40]

Run the shape against something concrete: a loan application, from arrival to decision. Day one, the workflow starts, verifies the applicant's data, and requests documents. Day three, the documents arrive, recorded as events delivered into the running workflow. Day four, the workflow requests a human underwriter review and sets a timer. Day nine, the review is recorded, the workflow orders a third-party appraisal, and sets another timer. Three weeks in, the appraisal returns, the decision step runs, the customer is notified, and the workflow closes. In between every one of those steps, machines rebooted, deployments shipped, instances died and were replaced. The workflow did not notice, because its state was never in any of those machines.

Temporal and Restate already provide durable execution; Restate also documents persistent agent sessions, approvals, and recovery controls.[41] Gas Town persists task state, hooks, agent identity, history, and orchestration state in Beads and Git-backed storage, so worker restarts do not erase everything either. The remaining design work is at the boundaries: which memory version belonged to the run, which external evidence must be retained, which dependencies can be rebuilt, and which credentials must be renewed. No workflow engine can restore a customer's changed record or make an expired grant current by replaying history. An orchestrator has to decide what is recorded for reconstruction and what is checked again before resuming.

This is the last infrastructure layer in the survey — the sections after it describe assembled agents, not components — and it is the layer that keeps state distinct from memory. Memory (Section 5.5) is what the system knows — facts, decisions, corrections. Durable orchestration is where the system is — which step, waiting on what, what happens next. A system can have one without the other; an orchestrated business process needs both. And durability is the floor of this layer, not the ceiling: once a system's state can be stood up at will, replaying a decision for an auditor, evaluating a system against its own history, and improving it over months rather than sessions all become buildable. Everything this book describes above this layer assumes it will exist.

5.9.1 Where durable orchestration shows up for an orchestrator

The encounters are decisions about what runs on this layer and what does not. Choosing the boundary: which parts of an orchestrated flow need durable guarantees (the approval wait, the payment step, the permit application) and which can stay ephemeral. Wiring the waits: the multi-day pause for human review is a durable timer, not a script that sleeps. And replaying: after a failure or an audit, the history is a first-class record of what step the workflow was on, what it had already done, and what

[40]Temporal's durability model per the company's documentation and engineering writeups: workflow state is durably recorded as an event history and reconstructed by replay on worker failure, with durable timers, signals delivered as recorded events, and at-least-once activity execution with idempotency mechanisms. Temporal, "What is Durable Execution," https://temporal.io/blog/what-is-durable-execution, and https://docs.temporal.io (including the workflow-determinism requirements: workflow code must be deterministic, and side effects belong in activities). The loan-application walkthrough is this book's illustration of the documented mechanics, not a reported case study.

[41]Temporal, "Durable AI," https://docs.temporal.io/ai (verified August 29, 2026) — workflows resume automatically after crashes, network timeouts, or multi-day waits for human approval. Restate, "AI / Agents," https://docs.restate.dev/ai (verified August 29, 2026) — durable execution, persistent sessions, approvals with pause and resume, cancel/kill/rollback for stuck agents.

it was waiting for, queryable step by step. The one discipline to know before adopting these engines: when a workflow is replayed to rebuild its state, the code has to make exactly the same decisions it made the first time. In Temporal, use workflow-safe APIs for time and randomness, and activities for external work such as model or service calls. Replay returns the recorded results; an activity that has not completed may be retried. External effects therefore need idempotency or deduplication, so resuming the work does not issue the same payment twice.

This layer also carries the audit burden asymmetry that Chapter 7 develops: operational work — the runs that touch customers, money, and production — is exactly the work that needs durability most, because it is the work whose interruption is expensive and whose record is demanded later. The question to ask of any long-running design: if this workflow dies at step seven, what is lost, what resumes, and who finds out?

5.10 Task agents

Task agents combine an inference engine with context, tools, state, and a loop — observe, interpret, choose, act, inspect the result, and then continue, hand off, request approval, or stop. A coding agent that reads a repository, edits a branch, runs tests, and opens a pull request is a task agent, and so is a research agent that gathers sources and returns an evidence packet. A classifier may choose a label with no tools at all, and an evaluator may inspect a result while being forbidden to change it. Because the word *agent* covers all of that, the contract matters considerably more than the label — which is the subject of the next section.

A note on the name, because the reader deserves the truth about it: "task agent" is this book's label, not the market's. The industry calls every one of these things an "agent" and is done with it, and I use "task agent" here as a working distinction rather than a settled product category.[42] The plain term is useful anyway, because the market's word is the problem the next section describes. What it points at is real: a bounded unit of delegated work, running inside some environment, against a contract. And to answer the question directly: a task agent is not a harness, and a harness is not a task agent. Hermes Agent, Goose, Claude Code, Cursor — those are harnesses and runtimes, the environments agents run inside (Section 5.11). The task agent is one bounded unit of work running inside one of them, the way a job is not the operating system it runs on.

5.10.1 A task agent, concretely

Strip the abstraction and a task agent has six parts: an objective (what result, for whom), the context it may read (files, records, search results), the tools it may call, the permissions bounding those calls, an output format, and stop conditions — what makes it finish, hand off, or ask a person. Those six parts are the contract, and the contract is what an orchestrator actually designs. Capability comes and goes with model releases; the contract is yours.

One unit of work, start to finish, as an illustrative Devin run. A developer assigns Devin, Cognition's cloud coding agent, a failing GitHub issue: the checkout page crashes when the cart is empty. Devin clones the repository, reproduces the crash, plans the fix, edits the files, runs the tests, and delivers a

[42]The industry term is simply "agent" or "AI agent" — see IBM, "What Are AI Agents?" https://www.ibm.com/think/topics/ai-agents, and Wikipedia, "AI agent," https://en.wikipedia.org/wiki/AI_agent, which notes there is no universally agreed definition; NIST's AI Agent Standards Initiative (announced August 2026) standardizes on "agent" as well. "Task agent" is this book's neutral label for a bounded unit of delegated work, chosen precisely because the market's word covers everything and therefore nothing.

pull request with the fix — then stops.[43] Nothing about that run is mysterious. The objective was the issue. The context was the repository. The tools were file edits and the test runner. The boundary — that it could open a pull request but not merge one — depends on repository permissions and explicit configuration. The output was the diff. The stop condition was the deliverable in hand; whether the tests actually pass is reported, not guaranteed. Every product in this section is a variation on that loop with different parts emphasized, and the orchestrator's job is knowing, for each one, where the six parts are written down.

5.10.2 The products

The current field, by what the agent's contract emphasizes:

- **Devin** (Cognition). The purest delegation: assign work through a web interface or Slack and get back a pull request. It runs in the cloud on managed infrastructure, billed by compute units rather than seats, because the product really is the agent rather than the tooling around it.
- **Claude Code** (Anthropic). The terminal-borne version: an agent that reads the repository, edits files, runs commands, and opens pull requests from the command line.
- **Codex** (OpenAI). Local and cloud coding work, including parallel tasks in isolated environments; a cloud task can deliver a branch and pull request for review.
- **GitHub Copilot's coding agent.** Issue-to-pull-request inside GitHub itself: assign an issue and the agent opens a draft PR with a working branch — the shortest path from ticket to code in the set, because it never leaves the platform the ticket lives on.
- **Cursor's agents.** Editor and cloud agents, including background runs that produce merge-ready pull requests; the platform drift around them is the next section's subject.
- **Deep research agents** (OpenAI, Google, Perplexity). The non-coding shape of the same idea: given a question, the agent searches, reads, cross-checks, and returns a cited report — an evidence packet instead of a diff, and a reminder that task agents are not a coding phenomenon.
- **OpenClaw.** The personal always-on version: wakes on a schedule, checks the things it was told to check, escalates what matters (Section 5.3.2 covered the economics of exactly this pattern).
- **The minimal end.** A classifier that labels a support ticket is one model call with a schema — also a task agent, with a tiny contract. An evaluator that critiques another system's output while being forbidden to change it — likewise. Small contracts are still contracts, and these are the two shapes most of an orchestrated system's volume is made of.

5.10.3 Where task agents show up for an orchestrator

Task agents are the units an orchestrator actually deploys, evaluates, and stops, so the encounter is the job itself. The recurring decisions: writing the contract, which is the task agent's real definition, not its label; evaluating — building the set of cases a task agent must pass before it runs unattended, and re-running that set when the model, the tools, or the instructions change; and supervising — the monitoring that notices a stuck loop, a budget overrun, or an agent quietly doing something adjacent to its task.

The ladder of Section 5.2.1 gives the placement rule: every task agent sits on one or more rungs, and the orchestrator's question is which rung this agent actually needs. Chapter 7 makes this the

[43] Product documentation: Devin, https://devin.ai; Claude Agent SDK, https://www.anthropic.com/news/agent-sdk; OpenClaw, https://docs.openclaw.ai; and the OpenAI and GitHub product documentation cited in the text. Comparative surveys: daily.dev, https://daily.dev/blog/best-ai-coding-agents-comparison, and Braintrust, https://www.braintrust.dev/articles/best-ai-coding-tools-2026. Capabilities are vendor claims; the six-part contract framing is this book's.

architecture question — which responsibilities become agents and which stay as tools — and Chapter 9 collects the failures. Every deployment still comes down to two questions: what is this agent's contract, and can the system prove it was honored?