

Quali Whitepaper

Building an Infrastructure Control Plane With AI: What It Really Takes

A build plan, a full cost model and an honest look at what AI can and cannot do.

By David Williams, Quali SVP, and Claude, Anthropic's AI model

Contents

Executive Summary.....	3
Part 1. Our Specification: The Infrastructure Control Plane Periodic Table.....	5
Part 2. The Build.....	8
Part 3. The Reality Check.....	15
Part 4. The Economics.....	21
Part 5. The Market.....	28
Part 6. What We Found.....	34
Sources.....	41

Executive Summary

Advances in AI code generation have been remarkable. AI keeps speeding up software development, and it lets people with little coding experience create working code. With AI, coding skill matters less and the ability to describe the outcome clearly matters more.

Production applications are different. Coding skill, a disciplined development process and governance remain critical. Even so, the progress has people asking how long it will be before enterprise platforms are built in-house instead of licensed, if AI can create code through a conversation. For now, this may be background noise, but the discussion is real and it is getting louder. That belief shapes build-versus-buy decisions worth tens of millions of dollars, yet no one had tested it with real numbers.

I recently published the Infrastructure Control Plane periodic table, introduced in [Nobody Agrees What an Infrastructure Control Plane Is. So, I Mapped It](#), to map the work a modern control plane has to do. It made an excellent design specification, so I put it to the test, to answer one question with numbers rather than opinions: what it now takes to build an infrastructure control plane with AI.

What I did. I treated that belief as a hypothesis and tested it as a build plan with Claude, Anthropic's AI model. Starting from zero, Claude and I planned an infrastructure control plane that would be production-ready, fully tested and competitive with what buyers can already purchase.

Our specification was the Infrastructure Control Plane periodic table, a vendor-neutral map of the work a modern control plane must perform. We used it as the backdrop for every decision: what to build, in what order, how to measure progress and how to judge whether the result was competitive. We assumed a skilled team using AI at every step. Claude acted as an engineering partner rather than a code generator, shaping the plan and stating plainly where AI would carry the load and where it would not.

We worked through the whole program: the order of work, the team, the failure modes, the testing calendar, the division of labor between AI and people, the biggest risks, every line of cost including AI and infrastructure, and how the market would move while we built.

What we found, in brief. AI made nearly every step faster and cheaper to write. It did not remove a single step. A production platform turned out to be a multi-year program for a sizable team, with most of the cost in people and much of the schedule spent proving the platform works rather than writing it. Two of our three conditions for done were achievable with enough time, people and money. The third proved far harder, for reasons that have nothing to do with how fast code is written. The build takes three years and \$18 million to \$26 million, and it launches three years behind, because it reaches today's market three years late. Keeping pace from there costs about \$44 million by year five and \$68 million by year seven, and the build is still three years behind. Drawing level on features costs about \$55 million by year five. Built without AI, the same platform would launch about eight months later and cost about \$3.7 million more.

And there is a gap no budget closes. Agentic platforms learn as they work, from every customer environment, every day. That learning is private and cumulative. A newcomer can rebuild the platform. It cannot rebuild what the platform has learned, and it starts learning only when it has customers of its own. Part 5 sets out why this changes what it means to catch up.

The final part of this document sets out the outcome in full: the numbers, what we would achieve, what we would not, and what it means for the idea that AI will simply do all of this.

What this estimate is not. It is the cost for an internal team to build the category baseline that the periodic table defines, assuming the plan runs as designed. It is not the cost or the value of any vendor's product, including Quali's. Mature platforms extend well beyond that baseline, with years of customer-driven hardening, integrations, partner work, certifications, products built on top and,

increasingly, what their AI agents have learned in production. A company also carries sales, support, partners and an installed base that no build budget buys.

How this document is organized

The document is long. However, it's not just a paper written by AI, it's a plan to create an Infrastructure Control Plane written **with** AI. The document follows the plan in six parts. Parts 1 to 5 build the case one layer at a time. Part 6 delivers the verdict. Each part opens with a short note on what it covers and closes with a key takeaway, so you can see where you are and skip ahead without losing the thread.

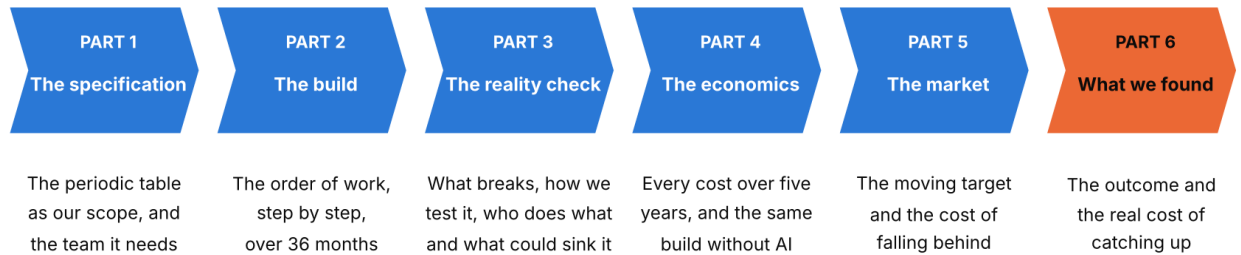


Figure 1. The journey through this paper.

Table 1. Sections of this paper.

Part	What it covers	What you will come away with
1 Our specification	What done means, the periodic table we built against and the team	The bar we set, how the table became our specification and the people needed to reach it
2 The build	Eleven build steps over 36 months, each mapped to elements of the table	Why order matters more than speed, where each step gets hard and how much of the table exists at each milestone
3 The reality check	Failure modes, testing and its calendar, the role of AI and people, the biggest risks	What AI changes, what it does not, and what could sink the program
4 The economics	Every cost over five years, including AI tokens, cloud, GPUs, lab, tooling, recruiting and audit, and the same build without AI	What it really costs to build and run, where the money goes and what AI saves
5 The market	Why the target moves while we build, the specification freeze, what being behind has cost others and the learning gap agentic AI creates	The variable most build-versus-buy plans leave out
6 What we found	The outcome, measured against our three conditions, the real cost of catching up, the price of staying behind and the risks of chasing a leader	What we would achieve, what we would not, and why "AI will do all this anyway" is only partly true

Part 1. Our Specification: The Infrastructure Control Plane Periodic Table

Every build needs a definition of what is being built. Ours was the [Infrastructure Control Plane periodic table](#), a vendor-neutral reference architecture that maps the work a modern control plane must absorb. It starts from the work, not from any product: discovering what exists, delivering governed environments on demand, operating them through change, drift and cost, retiring them, and governing the AI agents that increasingly act on infrastructure. That makes it a fair yardstick for a build and an honest one for a comparison.

The table organizes that work two ways.

1. Eight **domains** say what kind of capability each element is: Platform Core, Governance and Compliance, Developer Experience, Self-Service and Catalog, FinOps and Observability, Full Stack Infrastructure, AI and Agentic, and Integrations and API.
2. Four **pillars** say which outcome it serves: Curate (discover and inventory), Self-Service (deploy on demand), Operate (manage and optimize) and Agentic AI (governed AI operations).

Decomposed fully, the work comes to 104 elements. Each one carries a description of what it delivers and why it matters.

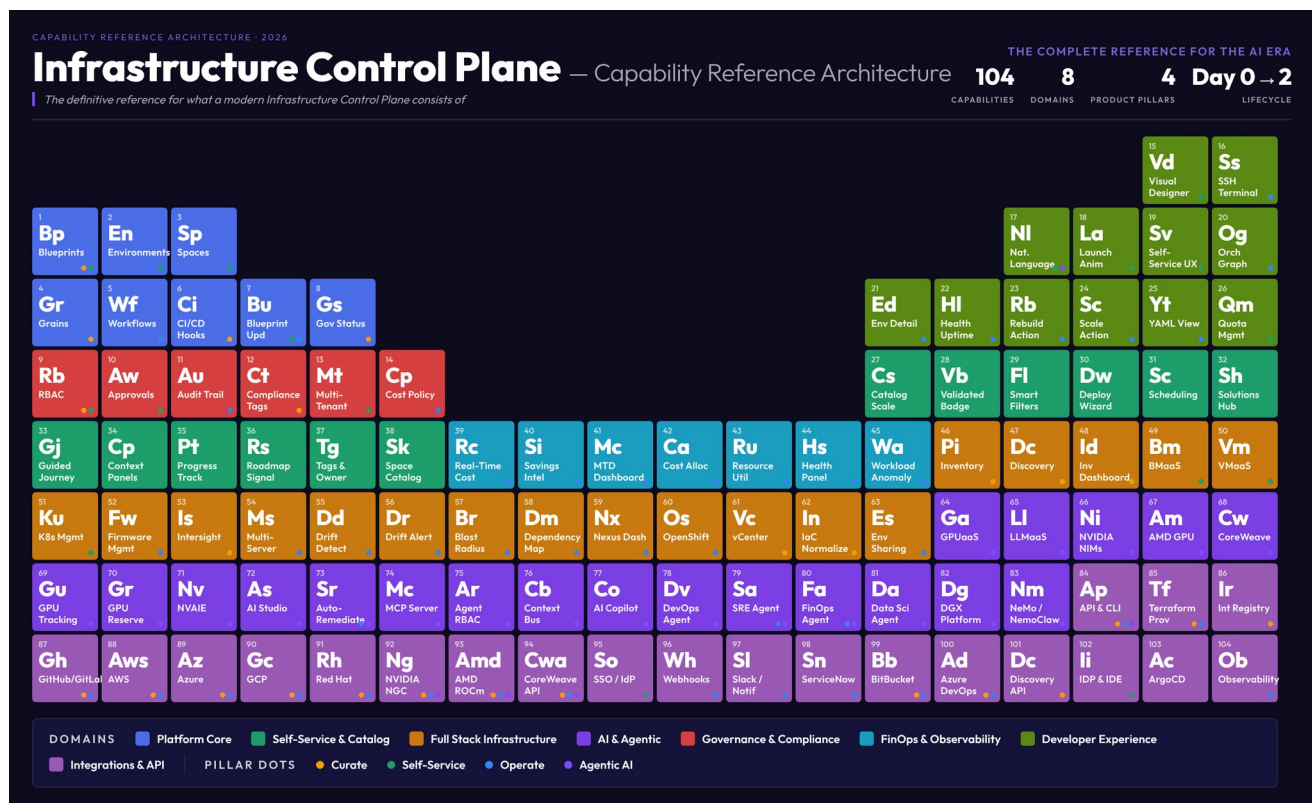


Figure 2. The Infrastructure Control Plane periodic table exactly as published on quali.com. 104 capabilities, colored by their eight domains, with dots marking the four pillars each one serves.

104 capabilities is not a target or a feature checklist. It is a receipt for work that most organizations already perform somewhere, in scripts, tickets, consoles and people's heads. For a build team, though, it is something more practical: a complete, written **statement of scope**.

How we turned the table into a build plan

Claude and I used the table in five ways, and it shaped every later part of this document.

- **As requirements.** Each element's "what it delivers" became a requirement, and its "why it matters" became the test of whether we had delivered it. An element counts as done only when it works and is tested.
- **As a dependency map.** The table makes the coupling visible. 19 elements serve more than one pillar, and the Operate pillar alone draws 46 elements from seven of the eight domains. Those shared elements had to be built first, because everything else stands on them.
- **As the build order.** We assigned every one of the 104 elements to one of our eleven build steps. Part 2 shows how much of the table is working at each milestone.
- **As a progress measure.** Counting tested elements at each milestone tells us how much of the specification exists at any point, which is more honest than counting lines of code.
- **As the competitive yardstick.** The table describes what buyers can already purchase today. Matching it at launch is our definition of competitive, and Part 5 explains why that target does not stand still.

Claude's first observation set the tone for everything that followed. No element in the table stands alone. The catalog, the cost engine, drift detection and the agents all read and write the same objects: environments, resources, owners, policies and versions. A demo can fake that shared model. A production platform lives or dies by it.

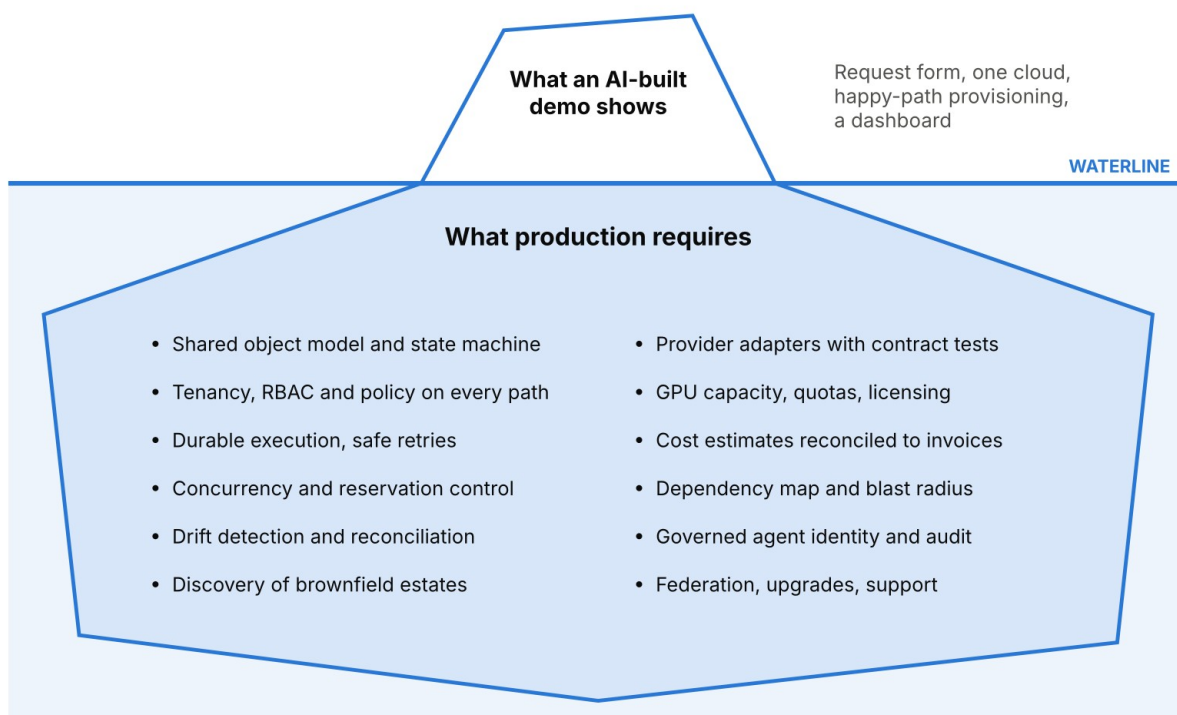


Figure 3. An AI-built demo covers the visible tip. Production requires everything below the waterline.

The team

Claude was direct about staffing. AI multiplies the output of skilled engineers. It does not remove the need for the skills. We sized a team around the disciplines the table demands and assumed every person uses AI tools throughout.

Table 2. Peak headcount by group.

Role group	Peak	Why we need it	Where Claude multiplies it
Architecture and product: principal architect, technical lead, product manager, designer, engineering manager	5	Owens the contracts and the irreversible decisions; decides what not to build	Drafts designs, generates edge cases that break them, writes decision records
Core platform engineering	6	Object model, workflow engine, state reconciliation, APIs, tenancy	Writes most first-draft code and unit tests once contracts exist
Integration and infrastructure	6	Cloud, virtualization, Kubernetes, bare metal, networking and GPU adapters	Drafts adapters from API documentation; humans prove them on real systems
Quality, security and reliability	5	Test automation, threat modeling, failure injection, on-call readiness	Generates test suites and failure scenarios; independent review stays human
Experience and documentation	2	Self-service interface, operator views, documentation	Builds interface components and first-draft documentation quickly

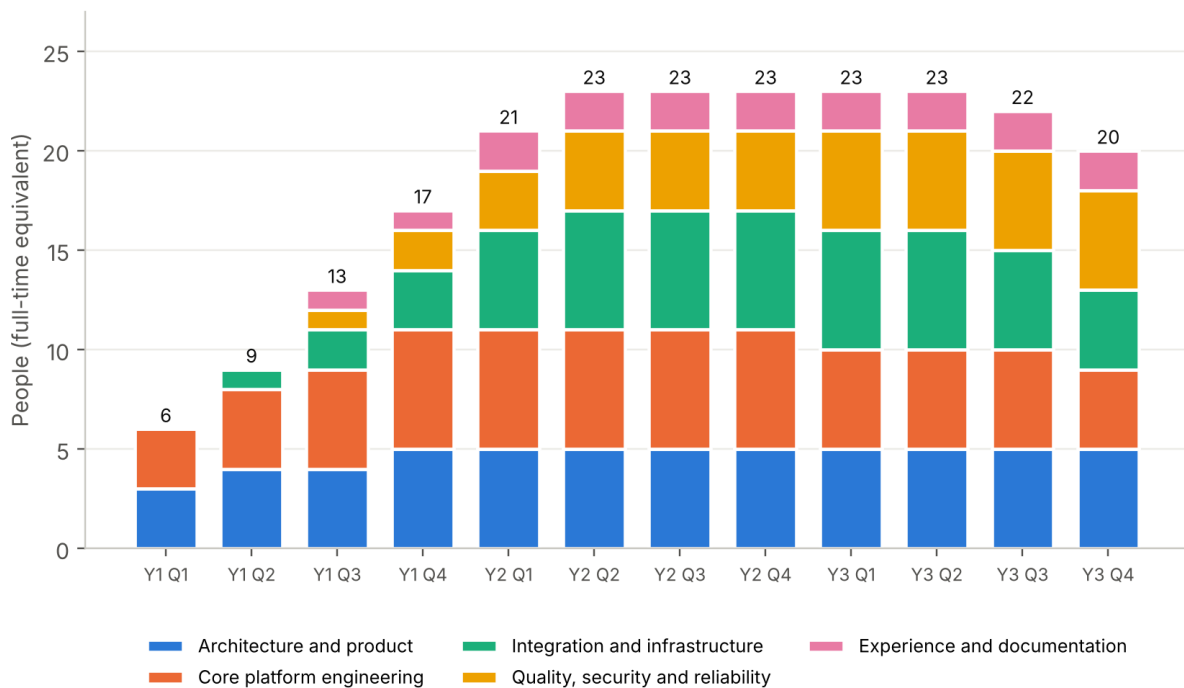


Figure 4. Team ramp by quarter. The team grows as foundations stabilize and integration work opens up, then shifts toward quality, security and reliability before general availability

Claude's take: Claude can multiply the output of every person in this table. It cannot replace the architect's judgment on decisions that are expensive to reverse, the integration engineer's access to a real switch or GPU node, or the on-call engineer's accountability at 2 a.m. A smaller team with heavy AI use is plausible. A team of zero is not.

Key takeaway: The periodic table gave us a complete, vendor-neutral specification: 104 elements across eight domains and four pillars. It is demanding because its elements share one model of environments, owners, policies and state. The plan needs 23 people at peak. AI multiplies each role; it does not replace any of them.

Part 2. The Build

In this part: the eleven steps of the build in the order we would take them, the elements of the table each step delivers, where each gets hard, what Claude contributes and how much of the table exists at each milestone.

The build order

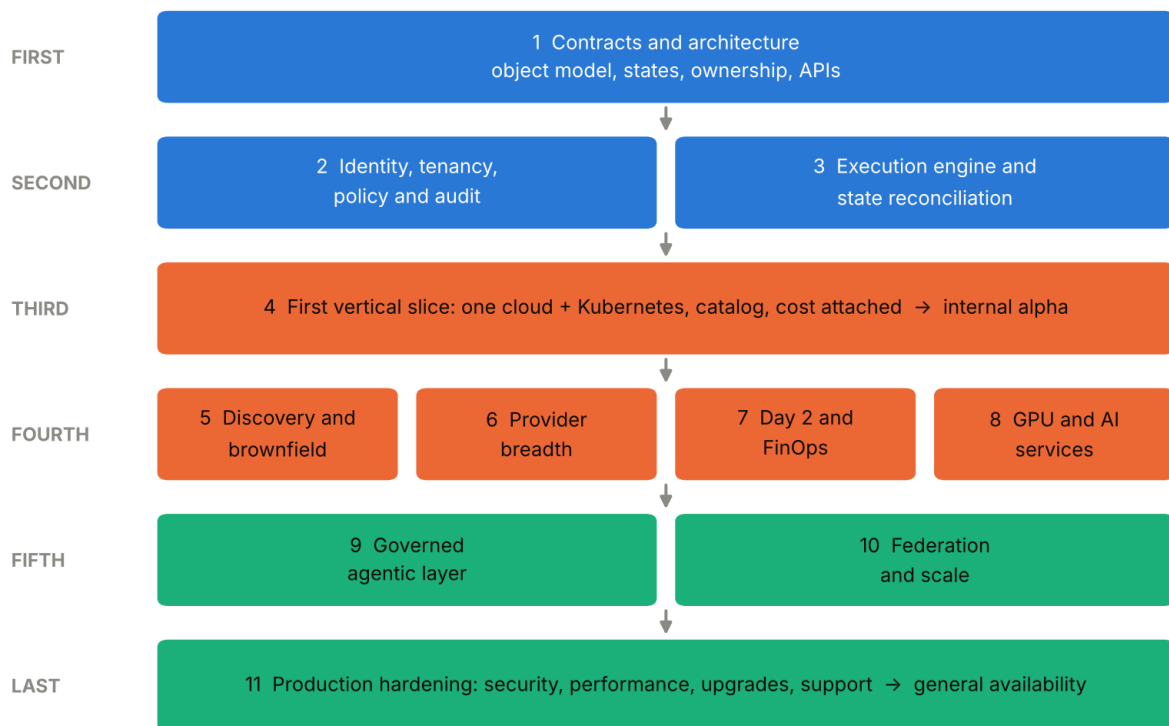
Order matters more than speed. Each layer depends on the contracts and foundations beneath it, and anything built out of order gets rebuilt. Claude and I settled on eleven steps across three years, with three milestones: an internal alpha at month 12, a design-partner beta at month 20 and general availability at month 36.



Figure 5. The table at the internal alpha, month 12. Lit elements are working and tested; faded elements are not yet built. The foundations come first: platform core, governance and the first slice of self-service, full stack and integrations.



Figure 6. The table at the design-partner beta, month 20: 54 of 104 elements. Only 4 of the 20 AI and agentic elements are lit and none of the agents, because agents are only as safe as the governance beneath them. All 104 arrive at launch in month 36.



Each layer depends on the contracts and foundations above it. Build out of order and you rebuild.

Figure 7. The dependency order. Foundations first, one complete slice second, breadth third, autonomy and federation only after the governance underneath them exists.

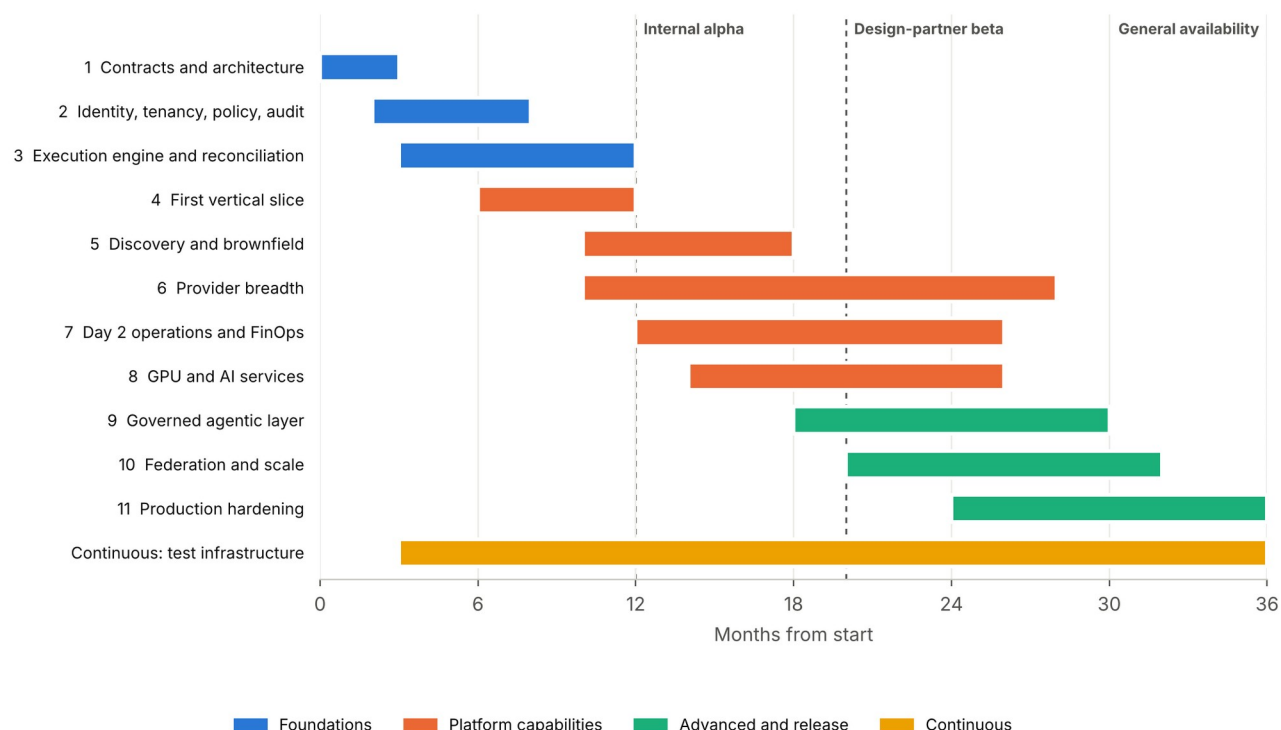


Figure 8. The 36-month roadmap. Steps overlap, but none starts before the layer it depends on is stable enough to build on.

Step 1: Define the contracts (months 0 to 3)

- **From the table.** No elements are delivered yet. This step designs the shared objects behind Blueprints (1), Environments (2), Spaces (3) and Grains (4), which every other element reads and writes.
- **What we build.** The object model and its rules. What an environment is, which resources belong to it, who owns it, which blueprint version created it, the states it moves through, desired versus observed state, API schemas and event meanings.
- **Why it comes first.** Every later capability reads and writes these objects. If the catalog, cost engine and agents each invent their own answer, we ship several products that share a login screen.
- **The hard part.** Making irreversible decisions with incomplete information. Whether an environment owns a shared network or only references it looks like a detail now. It decides how cost allocation, teardown and federation behave later.
- **Claude's role.** Drafts the specifications, proposes alternatives and generates scenarios designed to break them. Deciding which customer realities the model must serve is our call.

Step 2: Identity, tenancy, policy and audit (months 2 to 8)

- **From the table.** 8 elements: Spaces (3), RBAC (9), Approvals (10), Audit Trail (11), Compliance Tags (12), Multi-Tenant (13), Quota Mgmt (26), SSO / IdP (95).
- **What we build.** Single sign-on integration, role-based access at every layer, tenant isolation, hierarchical quotas, approvals and an audit trail.
- **Why it comes now.** Governance cannot be retrofitted. Every action path, whether interface, API, CLI, pipeline or agent, must pass the same checks from the first day.

- **The hard part.** Evaluating policy at the moment of action, not only at the moment of request. Handling revocation cleanly. Recording decisions in the audit trail without turning it into a store of exposed secrets.
- **Claude's role.** Implements policy plumbing fast and generates full permission-matrix tests. A security engineer owns the threat model.

Step 3: The execution engine and state reconciliation (months 3 to 12)

- **From the table.** 11 elements: Blueprints (1), Environments (2), Grains (4), Workflows (5), CI/CD Hooks (6), Orch Graph (20), Rebuild Action (23), Scale Action (24), Drift Detect (55), Drift Alert (56), API and CLI (84).
- **What we build.** Durable workflows, blueprint orchestration, safe retries, compensation for failed steps and a reconciliation loop that compares what should exist with what does. Kubernetes documents this controller pattern of moving current state toward desired state.¹
- **Why it comes now.** It is the engine every self-service, Day 2 and agentic capability will drive.
- **The hard part.** Partial failure. Temporal's documentation is blunt that activities must be idempotent because a worker can finish an action and crash before reporting success, so the action runs again.² Some infrastructure actions, such as deleting data or flashing firmware, cannot be undone at all. This is the longest foundation step for good reason.
- **Claude's role.** Strong at writing state machines and their tests. Subtle concurrency bugs across services surface only under deliberate fault injection, which we design together and run for real.

Step 4: The first vertical slice (months 6 to 12, internal alpha)

- **From the table.** 13 elements: Self-Service UX (19), Env Detail (21), YAML View (25), Catalog Scale (27), Smart Filters (29), Deploy Wizard (30), Scheduling (31), Tags and Owner (37), Space Catalog (38), Real-Time Cost (39), K8s Mgmt (51), GitHub/GitLab (87), AWS (88).
- **What we build.** One cloud plus Kubernetes, a small catalog and a complete self-service request from catalog to teardown, with ownership, cost and audit attached.
- **Why it comes now.** A thin, complete slice exposes gaps in the contracts while they are still cheap to fix.
- **The hard part.** Resisting breadth. Every stakeholder will want their cloud, their tool and their feature in the alpha.
- **Claude's role.** This is where AI moves fastest, because the contracts already exist to keep generated code consistent.

Step 5: Discovery and the brownfield estate (months 10 to 18)

- **From the table.** 7 elements: Gov Status (8), Inventory (46), Discovery (47), Inv Dashboard (48), IaC Normalize (62), Int Registry (86), Discovery API (101).
- **What we build.** Inventory across cloud accounts, virtualization and Kubernetes; detection of resources created outside any pipeline; normalization of existing Terraform, Helm and Ansible into governed blueprints.
- **The hard part.** Real estates are messy. Ownership is unknown, tags are missing, APIs are rate-limited, and discovery must never disturb production while it scans.
- **Claude's role.** Parses infrastructure code well and proposes blueprint wrappers. People validate the proposals against the live estate.

Step 6: Provider breadth (months 10 to 28)

- **From the table.** 20 elements: SSH Terminal (16), BMaaS (49), VMaaS (50), Firmware Mgmt (52), Intersight (53), Nexus Dash (59), OpenShift (60), vCenter (61), Terraform Prov (85), Azure (89), GCP (90), Red Hat (91), Webhooks (96), Slack / Notif (97), ServiceNow (98), BitBucket (99), Azure DevOps (100), IDP and IDE (102), ArgoCD (103), Observability (104).
- **What we build.** Additional clouds, virtualization, bare metal, data center networking, enterprise Kubernetes, ITSM, Git and CI/CD systems, identity providers and observability tools.
- **The hard part.** Every integration is a permanent commitment: authentication, pagination, throttling, eventual consistency and version changes, each needing test accounts or lab hardware to reproduce failures. This is the longest step on the roadmap.
- **Claude's role.** Drafts adapters quickly from API documentation. Documentation is often incomplete, and only contract tests against real systems reveal how an API actually behaves.

Step 7: Day 2 operations and FinOps (months 12 to 26)

- **From the table.** 12 elements: Blueprint Upd (7), Cost Policy (14), Health Uptime (22), Savings Intel (40), MTD Dashboard (41), Cost Alloc (42), Resource Util (43), Health Panel (44), Workload Anomaly (45), Blast Radius (57), Dependency Map (58), Env Sharing (63).
- **What we build.** Real-time cost, cost allocation, utilization, idle and anomaly detection, drift remediation, blueprint update management, dependency mapping and blast radius analysis.
- **The hard part.** Cost estimates differ from invoices that arrive later, and shared or on-premises capacity has no invoice at all. Dependency data is never complete, so blast radius analysis must show its own uncertainty rather than a confident picture built on gaps.
- **Claude's role.** Writes analytics and reporting code well. It cannot supply data that the estate does not expose.

Step 8: GPU and AI infrastructure services (months 14 to 26)

- **From the table.** 14 elements: GPUaaS (64), LLaaS (65), NVIDIA NIMs (66), AMD GPU (67), CoreWeave (68), GPU Tracking (69), GPU Reserve (70), NVAIE (71), AI Studio (72), DGX Platform (82), NeMo / NemoClaw (83), NVIDIA NGC (92), AMD ROCm (93), CoreWeave API (94).
- **What we build.** GPU inventory across vendors and GPU clouds, reservations, quotas, chargeback, model-serving deployments and software license validation.
- **The hard part.** Scarcity turns every allocation into a race. Driver, runtime and operator combinations must be validated on real hardware, and that hardware is expensive: on-demand H100 pricing currently runs from about \$2 to more than \$12 per GPU-hour depending on the provider.³
- **Claude's role.** Helps write scheduling and reservation logic. It cannot certify a driver stack on hardware it never touches.

Step 9: The governed agentic layer (months 18 to 30)

- **From the table.** 10 elements: Nat. Language (17), Auto-Remediate (73), MCP Server (74), Agent RBAC (75), Context Bus (76), AI Copilot (77), DevOps Agent (78), SRE Agent (79), FinOps Agent (80), Data Sci Agent (81).
- **What we build.** An MCP server exposing platform capabilities to agents, agent identities with scoped permissions, coordination between agents, specialist agents for operations and cost, approval paths and evaluation suites.
- **Why it comes this late.** Agents are only as safe as the identity, policy, state and audit beneath them. Built first, they would be ungoverned automation with an API key.

- **The hard part.** Hostile text arriving through tool outputs, two agents acting on the same resource, decisions made on stale observations and proving that agents stay inside their authority. Gartner predicts over 40% of agentic AI projects will be canceled by the end of 2027, citing escalating costs, unclear value and inadequate risk controls.⁴

Claude's take: *This is where it is useful and most dangerous. Claude can build agents quickly. Proving they stay inside their authority under adversarial input takes evaluation suites, red-teaming and people who decide how much autonomy is acceptable. Skip that work and this layer becomes the risk it was meant to control.*

Step 10: Federation and scale (months 20 to 32)

- **From the table.** 1 element: Multi-Server (54).
- **What we build.** Multi-site management, regional authority, operation through network partitions, high availability and performance across thousands of environments.
- **The hard part.** Distributed authority. When a site reconnects after operating alone, the platform must decide which record wins, which policy version applied and which commands have expired, and it must keep the evidence of conflicts instead of silently overwriting it.
- **Claude's role.** Helps design the protocols and generates partition scenarios. Confidence comes only from running those scenarios over time.

Step 11: Production hardening (months 24 to 36, general availability)

- **From the table.** 8 elements: Visual Designer (15), Launch Anim (18), Validated Badge (28), Solutions Hub (32), Guided Journey (33), Context Panels (34), Progress Track (35), Roadmap Signal (36). These are onboarding and catalog-trust elements that only make sense once the platform is proven.
- **What we build.** Independent penetration testing, compliance attestation, performance tuning, upgrade and migration paths, runbooks, documentation and a support model.
- **The hard part.** This step produces evidence, not features, and evidence takes calendar time. A SOC 2 Type II report, for example, examines controls over an observation period that commonly runs from three to twelve months.⁵ AI does not shorten an audit window.

How much of the table exists, and when

Counting tested elements against the table shows how the platform fills in. The foundations deliver few visible elements but carry everything that follows. The slope steepens once breadth work begins, and the agentic pillar arrives last because it depends on the governance underneath it.

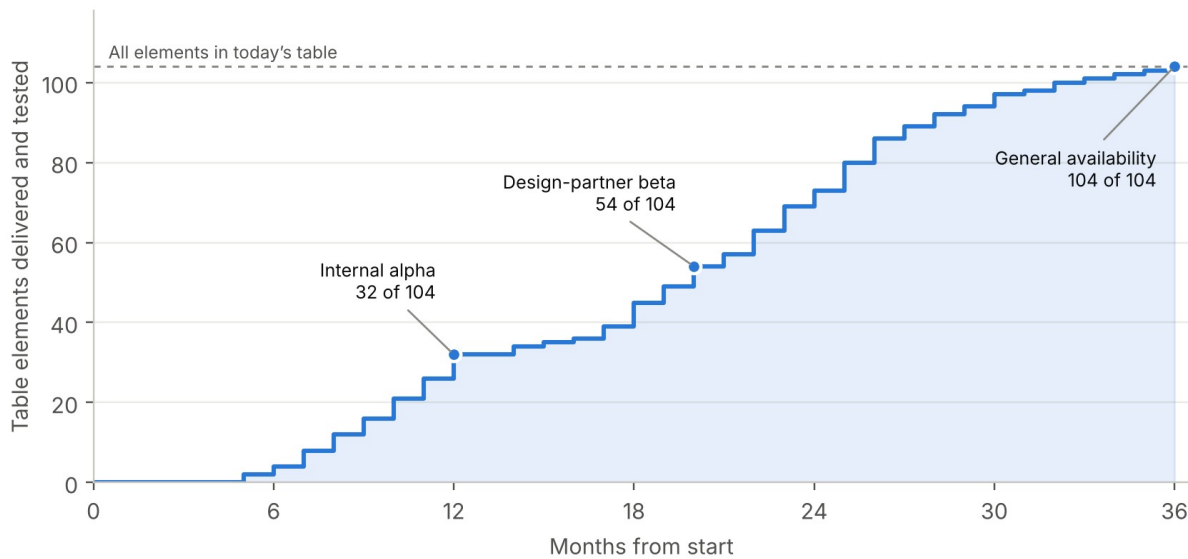


Figure 9. Elements of today's table delivered and tested, month by month. The foundations take a year to produce a third of the table; the rest arrives once they are stable.

Table 3. An element counts once it works end to end and is tested. Elements that serve several pillars count in each.

Pillar	Internal alpha (month 12)	Beta (month 20)	Launch (month 36)
Curate (26 elements)	8	16	26
Self-Service (29 elements)	16	19	29
Operate (46 elements)	13	21	46
Agentic AI (25 elements)	1	5	25
All elements	32 of 104	54 of 104	104 of 104

Key takeaway: Order matters more than speed. Contracts, governance and the execution engine come first; agents and federation come last. Measured against the table, we would have 32 of 104 elements at the internal alpha, 54 at the beta and all 104 only at launch.

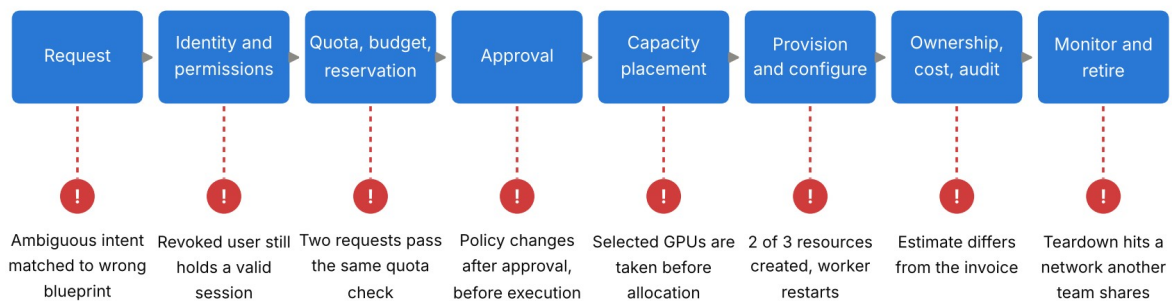
Part 3. The Reality Check

In this part: *what breaks in a single request, what fully tested means and how long it takes, how work divides between AI and people, and the challenges that could sink the program.*

One request, eight ways to fail

To pressure-test the plan, Claude and I traced a single request through it: a data scientist asks for a four-GPU training environment for 48 hours with a \$200 cost cap. The user sees one request and one running environment. Mapped against the table, that one request draws on 23 elements from all eight domains, from AI Copilot (77) and RBAC (9) to GPU Reserve (70), Cost Policy (14) and Drift Detect (55). It passes through eight stages, and each has a failure mode that a demo never encounters.

What the user sees: one request, one running environment



Failure modes (critical) that a demo never encounters and production must handle

Figure 10. One self-service GPU request, and the failure each stage must handle. Every one of these requires a designed behavior and a test that proves it.

Consider the sixth. The provider creates two of three resources, and the worker restarts before it records the second. A naive retry allocates duplicate GPUs. A naive cancellation deletes a network that another team shares. Handling it correctly requires stable operation identifiers, provider-specific recovery rules and reconciliation against what actually exists. That is one failure mode in one request.

What “fully tested” means

Generated code arrives with generated tests, and generated tests tend to confirm what the code already does. Production confidence comes from test layers that prove behavior against reality.

Table 4. The test layers, what each proves and what Claude contributes.

Test layer	What it proves	What Claude contributes
Unit and property tests	Logic holds across generated inputs	Writes most of them, quickly
Provider contract tests	Adapters match how real APIs behave	Drafts them; real accounts and hardware prove them
End-to-end lifecycle tests	Request to teardown works across every component	Drafts scenarios; requires live environments
Failure injection	Restarts, partial failures and partitions are handled safely	Designs scenarios; engineers judge outcomes
Concurrency and scale	No double allocation; performance holds at volume	Generates load; requires infrastructure to run it
Security testing	Tenant isolation and privilege boundaries hold	Assists; independent testers are required
Agent evaluations	Agents stay inside their authority under adversarial input	Builds suites; people set acceptance thresholds
Upgrade and migration	Customers move between versions without losing state	Drafts paths; requires production-like data

How long it takes to be fully tested. Code can be written faster with AI. Evidence cannot. Some testing runs continuously from the first months. The tests that prove the platform is safe to trust depend on real users, real hardware and independent reviewers, and they run on a calendar of their own.

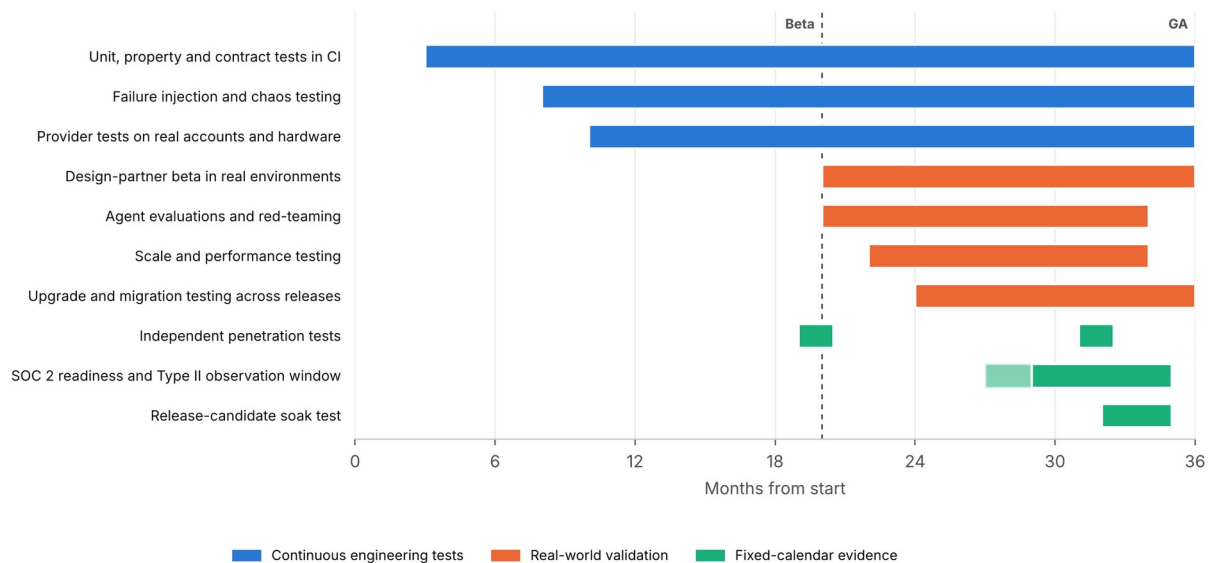


Figure 11. The testing calendar. Continuous engineering tests start early. Real-world validation starts at the beta. Independent evidence, such as penetration tests, the SOC 2 observation window and a release-candidate soak test, runs on fixed calendars that AI cannot compress.

In our plan, the evidence calendar runs from the beta at month 20 to launch at month 36. It includes 16 months of design-partner use, scale and performance testing, agent red-teaming, upgrade testing across successive releases, two independent penetration tests, a six-month SOC 2 Type II observation window⁵ and a 90-day soak test of the release candidate. Even if every line of code were finished on the first day of the beta, launch would still depend on that calendar.

Test infrastructure is also a standing cost, covered in full in the next section: cloud accounts for every provider we support, GPU capacity, a bare-metal and networking lab and environments for upgrade testing.

Who does what: the role of AI and the role of people

The most useful thing the exercise produced was a clear division of labor. AI is real leverage, and our plan assumes it everywhere. Google's 2025 DORA research found that close to 90% of technology professionals use AI at work and more than 80% believe it has made them more productive. AI adoption is now associated with higher delivery throughput, while software delivery instability continues to rise. DORA describes AI as an amplifier that magnifies the strengths of high-performing organizations and the dysfunctions of struggling ones.⁶ Developers remain cautious: in Stack Overflow's 2025 survey, 46% said they distrust the accuracy of AI tools and only 3% reported high trust.⁷

Claude and I mapped where that leverage lands across the eleven steps. The pattern is consistent. AI leverage is highest where the work is writing code inside clear contracts. It falls as the work moves toward proving behavior on real systems, and it reaches zero at the point of decision and accountability.

Nothing works the first time. Our plan assumes AI produces good work efficiently, inside a disciplined loop. In practice, AI rarely produces the right result on the first pass. This paper took more than a dozen drafts with Claude, each one checked, corrected and rerun by a person: figures recalculated, diagrams redrawn and claims tightened. A platform build repeats that loop across thousands of decisions, and every pass needs someone who knows what right looks like. The plan budgets for the loop running well. Where it runs badly, schedules slip, and that is where the overruns in Part 4 come from.

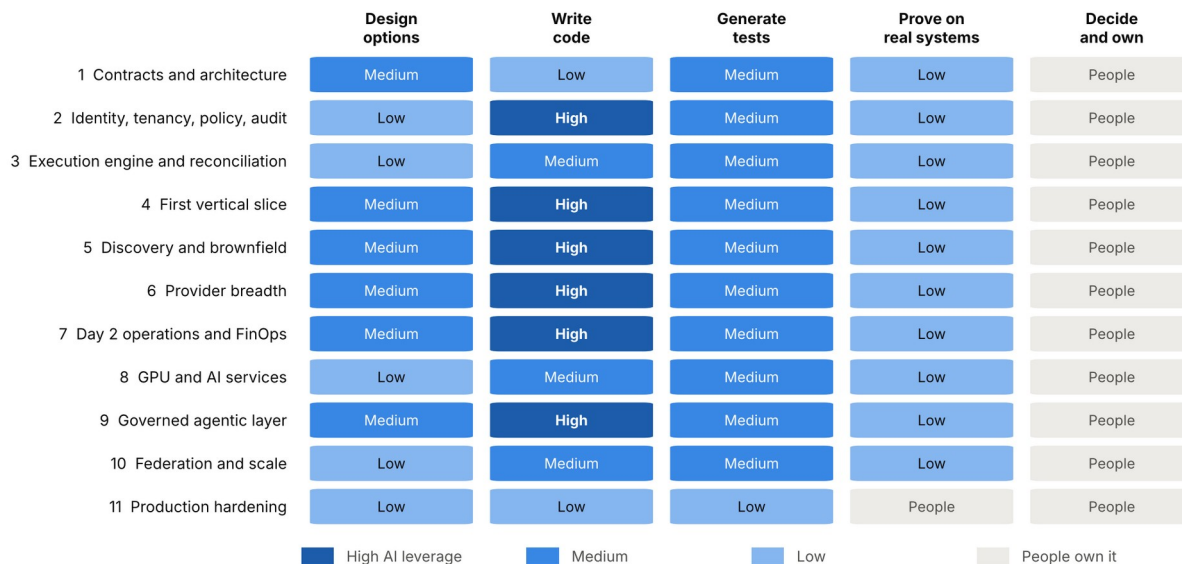


Figure 12. AI leverage by build step and activity, as Claude and I assessed it. Writing code is where AI carries the most load. Deciding and owning the result stays with people at every step.

Table 5. The role of AI and the role of people, by area of work.

Area	The role of AI	The role of people
Architecture	Proposes options, finds contradictions between specifications, drafts decision records, generates scenarios that break a design	Choose among options, make the irreversible decisions, decide which customer realities the platform must serve
Code	Writes most first drafts, refactors, migrations and boilerplate; keeps generated code consistent with the contracts it is given	Set standards, review every change to critical paths, and keep enough understanding of the system to change it safely
Testing	Generates unit and property tests, failure scenarios, load scripts and agent evaluation suites	Define what passing means, run tests on real accounts and hardware, judge ambiguous results
Integrations	Drafts adapters from API documentation and translates between provider models	Hold the accounts and lab hardware, discover how APIs actually behave, work with vendors when they change
Security	Scans code, drafts threat models, flags risky patterns	Own the threat model, commission independent testing, sign off on release
AI agents	Builds agents, tools and evaluation harnesses	Set the limits of autonomy, red-team agent behavior, approve which actions run without a person
Operations	Triages alerts, summarizes incidents, proposes fixes	Hold the pager, make customer-facing calls, decide when to roll back
Product	Synthesizes feedback, drafts documentation and release notes	Talk to customers, set priorities, decide what not to build

Four working rules came out of that mapping:

1. **AI proposes, people decide.** Every irreversible choice has a named human owner.
2. **Contracts before code.** AI is most productive inside tight specifications. Without them, it generates fast, plausible and inconsistent code.
3. **Every AI output meets a human-owned test.** Generated tests tend to confirm what generated code already does, so acceptance criteria come from people.
4. **Comprehension is a deliverable.** For every critical path, at least one engineer must understand the code well enough to diagnose it during an incident, whoever wrote it.

Claude's take: *Claude compresses the distance from a clear decision to working code. It does not supply the decision, the production history that tells you which edge cases matter, the hardware to test on or the accountability for a release. In this plan it carries the most load in steps 4 through 7, where contracts exist and feedback is fast, and Claude is least able to stand alone in steps 1, 8 and 11, where the work depends on judgment, real hardware and independent evidence.*

The biggest challenges, and what happens if we get them wrong

Claude and I then ranked the challenges that could sink the program. Some are technical, some are organizational, and one has nothing to do with our code at all. Each carries a specific ramification if it is not managed deliberately.

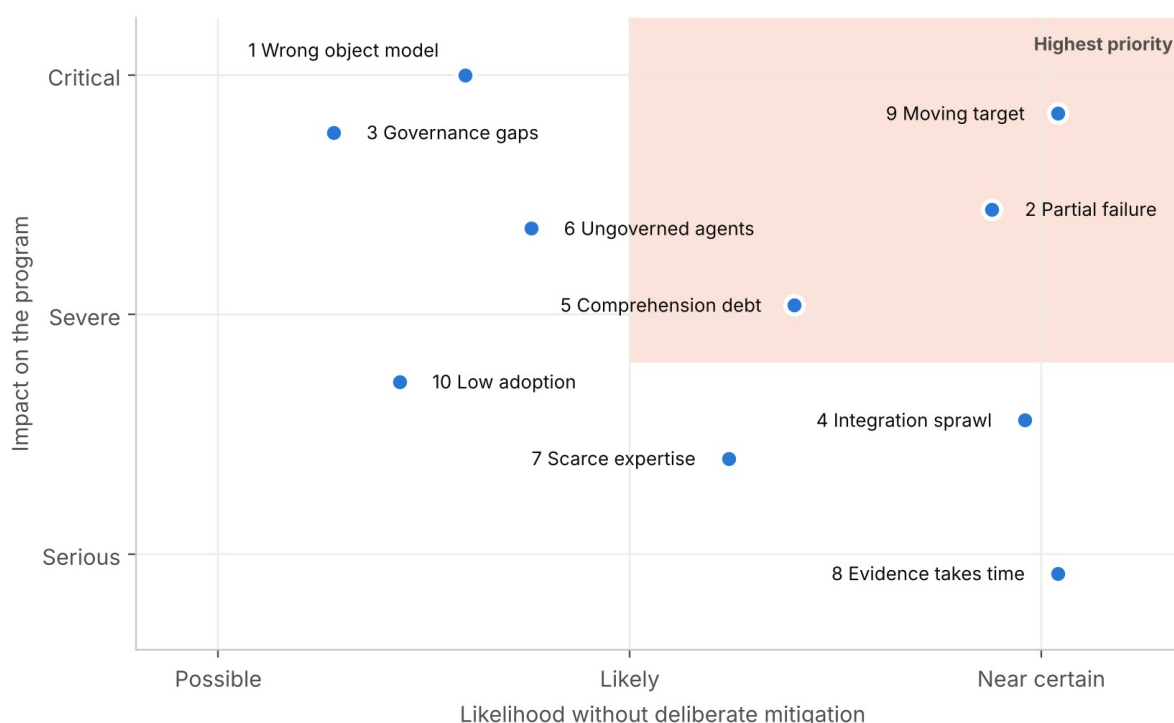


Figure 13. Our assessment of the ten biggest challenges by likelihood and impact if left unmanaged. Partial failure, the moving target and comprehension debt sit in the highest-priority zone.

Table 6. The ten biggest challenges, why each is hard and what happens if we get it wrong.

Challenge	Why it is hard	Ramification if we get it wrong
1 Wrong object model	Foundational decisions are made before customer reality is fully known	Errors spread into cost allocation, drift, teardown and federation; fixing them means rebuilding layers that customers already depend on
2 Partial failure and concurrency	Infrastructure APIs fail halfway, slowly and inconsistently	Duplicate GPUs, orphaned spend and deleted shared resources; one serious incident can end trust in self-service
3 Governance gaps	Pressure to demo features before tenancy and policy are complete	Cross-tenant exposure and failed security reviews; enterprise buyers stop at the security questionnaire
4 Integration sprawl	Every provider changes on its own schedule	Maintenance consumes the roadmap; a provider API change breaks customers before we notice
5 Comprehension debt	AI produces code faster than people can understand it	Incidents that nobody can diagnose quickly, rising instability and code review that becomes the bottleneck
6 Ungoverned agents	Agents are easy to build and hard to bound	An agent acts beyond its authority on production infrastructure; the program joins the agentic projects Gartner expects to be canceled
7 Scarce expertise	Distributed systems, infrastructure integration and GPU skills are rare	Hiring delays move every milestone, and knowledge concentrates in a few people the program cannot afford to lose
8 Evidence takes time	Audits, soak tests and design-partner validation cannot be rushed	General availability slips regardless of how fast code is written
9 The moving target	The market keeps improving while we build	We ship at parity with the past, and the budget grows to chase a line that keeps moving

Challenge	Why it is hard	Ramification if we get it wrong
10 Low adoption	Users already have workarounds, scripts and habits	A finished platform that few people use, which turns the entire investment into cost

Three of these deserve particular attention because they compound. Partial failure is near certain in infrastructure, and every unhandled case becomes a customer incident. Comprehension debt is new to the AI era: the faster code arrives, the more deliberate the team must be about understanding it, or the platform becomes something nobody can safely change. And the moving target sits outside our control entirely, which is why it gets its own section below.

Key takeaway: AI carries the most load in writing code. People own the decisions, the proof on real systems and the accountability. The biggest risks, partial failure, comprehension debt and the moving target, are not solved by writing code faster.

Part 4. The Economics

In this part: every cost line, quarter by quarter, through the build and two years of operation, including the AI itself and the infrastructure we build and test on, and what the same build would take without AI.

The full bill: people, AI, infrastructure and assurance

A credible plan has to count everything, including the AI itself and the infrastructure the team builds and tests on. We modeled every cost line quarter by quarter, through the three-year build and two years of running the platform afterward. Where a published price exists, we used it. Where the cost depends on our choices, we stated the assumption and gave a range.

People. The U.S. Bureau of Labor Statistics puts the median software developer wage at \$135,980.⁸ Benefits account for 30% of private employer compensation costs,⁹ which lifts that median developer to about \$194,000 before equipment, management and facilities. The engineers this plan needs earn well above the median, so we used \$250,000 per fully loaded person-year. After launch, a sustaining team of about 12 people keeps the platform current.

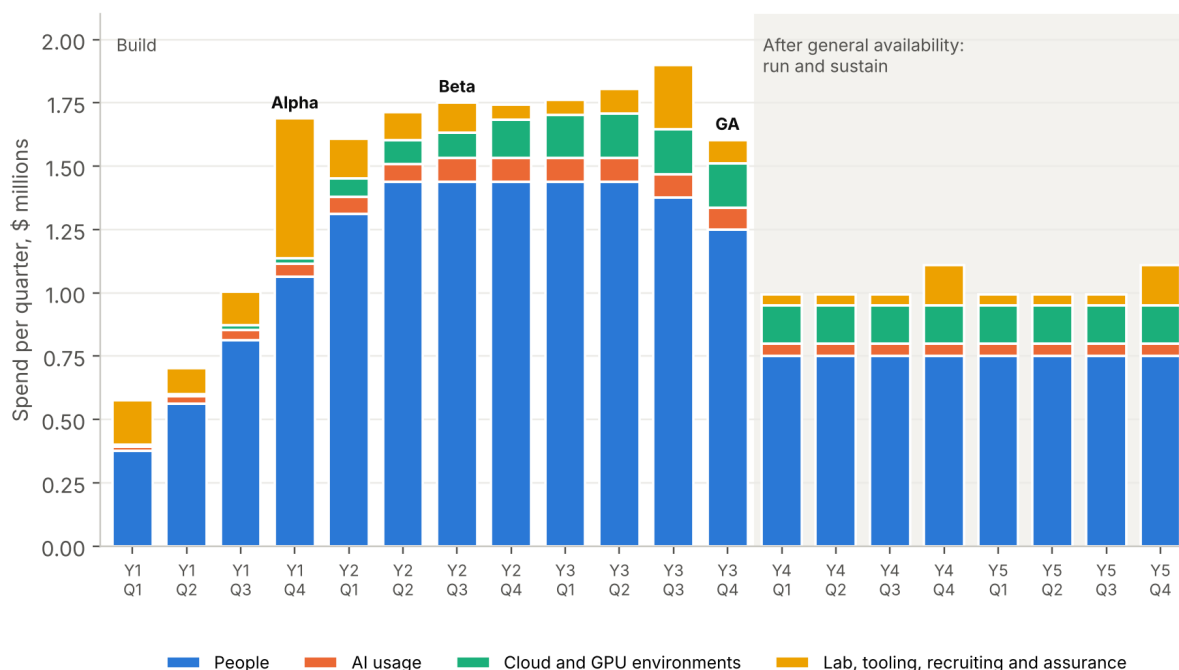


Figure 14. Estimated spend per quarter, mid case, through launch and two years of operation. People dominate every quarter. Spend falls after launch but does not stop.

Table 7. The cost of the build and of each year after launch, by cost line.

Cost line	Build, 3 years (mid)	Build range	Per year after launch
People	\$13.9M	\$11.3M to \$17.8M	\$3.0M
AI for engineering: coding assistants and agents	\$0.67M	\$0.17M to \$1.67M	\$0.14M
AI in the pipeline: automated review, test generation, agent evaluations	\$0.17M	\$0.05M to \$0.62M	\$0.05M
Cloud environments: development, CI, integration, staging, multi-region, backup, observability, design-partner hosting	\$0.95M	\$0.67M to \$1.52M	\$0.51M
GPU test capacity	\$0.23M	\$0.14M to \$0.55M	\$0.10M
Lab hardware, tooling, licenses, recruiting, security testing and audit	\$1.90M	\$1.07M to \$3.05M	\$0.29M
Total	\$17.9M	\$13.3M to \$25.2M	\$4.1M

The people range reflects scope assumptions of 45 to 71 person-years. Other ranges reflect usage and price assumptions described below.

What drives each non-labor line

Table 8. How we estimated each non-labor cost line.

Cost line	How we estimated it	Basis
AI for engineering	Anthropic reports that enterprise Claude Code use averages about \$13 per developer per active day, or \$150 to \$250 a month, with 90% of users below \$30 per active day. ¹⁰ Our plan runs several agents in parallel, and agent teams can use about seven times the tokens of a standard session. We budgeted \$1,000 per person per month, with a range of \$250 to \$2,500.	Published usage data; heavier use is our assumption
Token prices	Opus 5.5 costs \$4 per million input tokens and \$20 per million output tokens. Sonnet 5.5 costs \$2 and \$10. Cached context is read at 5% of the input price, and batch jobs cost half. ¹¹	Published list prices
AI in the pipeline	Automated review and test generation on every change, plus agent evaluations from month 18: about 10,000 evaluation runs a week, each roughly 60,000 input and 6,000 output tokens on Sonnet 5.5. That is about \$0.18 a run, or \$23,400 a quarter.	Published prices; volumes are our assumption
Cloud environments	Per-engineer development environments, CI, a persistent test cluster for each supported platform plus short-lived test resources, staging, a multi-region federation test bed, backups, observability and hosting for design partners: about \$47,000 a month at peak. One persistent test cluster costs about \$956 a month: \$0.10 an hour for the managed Kubernetes control plane ¹² plus six 4-vCPU instances at \$0.2016 an hour. ¹³ Large CI runners cost \$0.022 a minute. ¹⁴ Infrastructure monitoring runs \$15 to \$23 per host per month and logs \$0.10 per GB. ¹⁵	Published prices; environment sizes are our assumption
GPU test capacity	One 8-GPU NVIDIA H100 node 40 hours a week and one 8-GPU AMD MI300X node 20 hours a week from month 12, plus scale-test bursts. On-demand H100 rates run from \$1.99 to \$12.29 per GPU-hour, ³ and MI300X from \$2.59 to \$6.00. ¹⁶	Published prices; hours are our assumption
Lab hardware	Bare-metal servers, data center switches and GPU systems that cannot be rented by the hour for every test: \$250,000 to \$600,000 up front, plus \$2,000 to \$6,000 a month for space, power and support.	Our assumption; requires vendor quotes
Tooling, licenses and recruiting	Developer, security and design tooling at \$1,500 to \$4,000 per person a year; \$30,000 to \$120,000 a year in test licenses for third-party platforms; \$15,000 to \$40,000 per hire across about 30 hires, including turnover.	Our assumption
Security testing and audit	A full product security assessment typically costs \$25,000 to \$100,000 or more, and a red-team exercise \$50,000 to \$150,000 or more. ¹⁷ A first-year SOC 2 Type II program runs about \$50,000 to \$170,000 including readiness, tooling and the audit. ¹⁸ We budgeted two assessments, one agent red team and SOC 2 before launch.	Published ranges

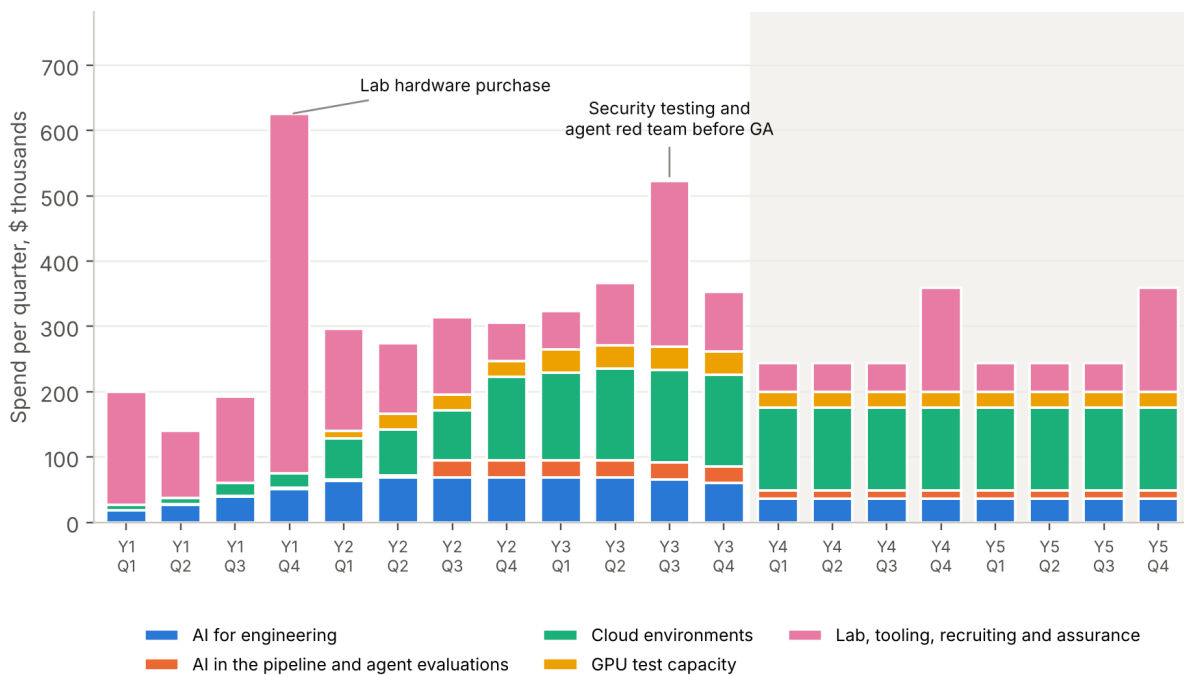


Figure 15. Everything except people, per quarter. Cloud environments grow as provider coverage widens. The two spikes are the lab purchase and pre-launch security testing. AI usage is a steady but small line.

Measured against the table, the build costs about \$172,000 per element on average, before the platform has a single production customer. Four observations stand out. First, people are 78% of the build. Second, AI usage, including every token for coding, review and agent testing, is under 5%. It is a real cost, and in the heavy-usage case it passes \$2 million, but it is not what makes this expensive. Third, the platform keeps costing money after launch: about \$4.1 million a year to run, maintain and improve. Fourth, an AI-assisted prototype covering the happy path of the GPU request, three to five engineers for two to three months, costs roughly \$125,000 to \$315,000.

The demo is less than 2 percent of the bill.

History warns that even this estimate is optimistic. In a study of more than 5,400 IT projects, McKinsey and the University of Oxford found that projects with an initial budget above \$15 million ran 45% over budget on average and delivered 56% less value than predicted, with each additional year of duration adding 15% to the overrun.¹⁹ A three-year, \$18 million platform build sits squarely in that territory.

The plan assumes a smooth run. Every estimate in this part assumes the program runs as designed. It never does. The same study found large projects ran 7% over time on average.¹⁹ Applied to this build, a typical overrun puts the cost nearer \$26 million, in a range of about \$19 million to \$37 million, and launch at about month 39. We keep the smooth-run mid case as the base for every comparison that follows, so the numbers stay comparable, but the realistic figure is higher. Treat \$17.9 million as the entry ticket, not the price of competing.

The same build, with and without AI

Every number so far assumes AI at every step. To show what that assumption is worth, Claude and I priced the same platform, with the same scope and the same quality bar, built the way teams worked before AI coding tools.

The evidence on AI's effect is mixed, so we used it with care. In a randomized trial at Google, 96 engineers finished an enterprise coding task about 21% faster with AI features.²⁰ Field experiments with 4,867 developers at Microsoft, Accenture and a Fortune 100 company found a 26% increase in

completed tasks.²¹ McKinsey found developers wrote new code in nearly half the time and documented it in about half the time, but gains fell below 10% on highly complex tasks.²² And in METR's 2025 trial, 16 experienced open-source developers took 19% longer with AI on code they knew well, while believing it had made them 20% faster.²³ AI saves the most on well-specified, routine work and the least where judgment and deep context dominate.

We applied that pattern activity by activity. AI saves the most time writing code, tests and documentation, some in design, little in proving the platform on real systems and none in decisions and accountability. We assumed the same hiring ramp and the same peak team in both cases, so the extra work stretches the schedule rather than the headcount. The 16-month evidence calendar runs the same either way.

Table 9. Share of the work and the time AI saves, by activity.

Activity	Share of the work, with AI	Time AI saves (low / mid / high)	Basis
Design and specification	14%	10% / 20% / 30%	Drafting options and the scenarios that break them; architects still choose
Writing code	27%	20% / 35% / 50%	Google trial about 21% faster; field experiments 26% more tasks; McKinsey nearly half the time on new code
Tests, scenarios and documentation	15%	25% / 40% / 55%	McKinsey about half the time on documentation; generated tests need human acceptance criteria
Proving on real systems	29%	5% / 10% / 15%	Real accounts, hardware and partners set the pace
Decisions, review and coordination	15%	0% / 0% / 5%	Accountability does not transfer

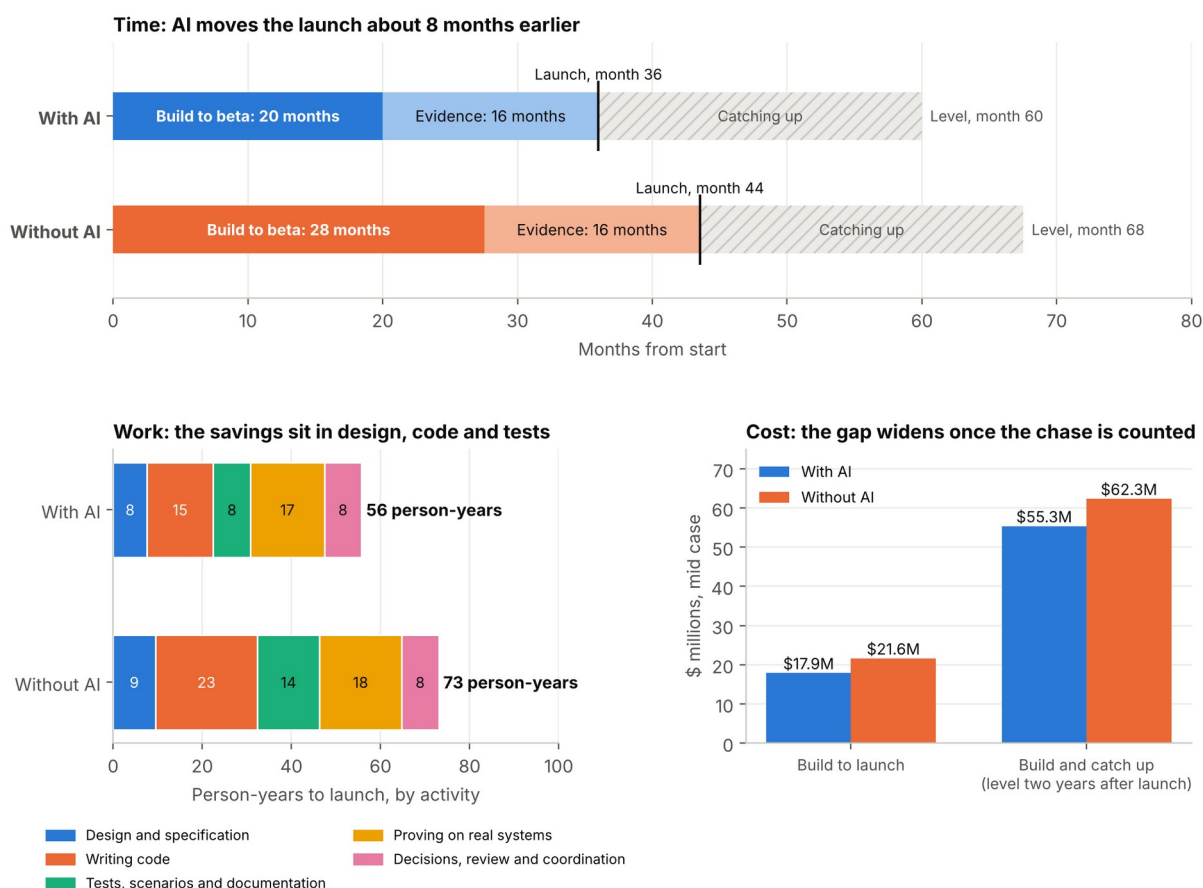


Figure 16. Mid case. Top: without AI, the build reaches its beta at month 28 instead of month 20, and launch moves from month 36 to month 44; drawing level with the market moves from month 60 to month 68. Bottom left: the extra 17 person-years sit almost entirely in design, code, tests and documentation. Bottom right: the build costs \$3.7 million more without AI, and the difference grows to \$7 million once the chase after launch is counted.

Without AI, the same platform takes about 73 person-years instead of 56. The beta moves from month 20 to month 28 and launch from month 36 to month 44. The build costs about \$21.6 million instead of \$17.9 million, \$3.7 million or 21% more. AI usage costs about \$0.8 million, so in the mid case every dollar spent on AI saves about \$5 of build cost. Across our range, building without AI delays launch by 4 to 14 months and adds \$1.3 million to \$7.6 million.

Table 10. The same build with and without AI.

Measure	With AI	Without AI (mid)	Without AI (range)
Person-years to launch	56	73	64 to 87
Design-partner beta	Month 20	Month 28	Month 24 to 34
Launch	Month 36	Month 44	Month 40 to 50
Cost to launch	\$17.9 million	\$21.6 million	\$19.2 million to \$25.4 million
Of which AI usage	\$0.8 million	None	None
Gap at launch, market improving 20% a year	73 points	94 points	82 to 113 points
Total to draw level two years after launch	\$55.3 million	\$62.3 million	\$58.1 million to \$69.1 million

The months matter more than the money. Eight months later is eight more months of market movement. At 20% a year, a launch at month 44 arrives about 94 points behind instead of 73, and drawing level two @2026 Quali. All rights reserved.

years after launch costs about \$62 million in total instead of \$55 million. AI's largest contribution to this plan is time to market, and Parts 5 and 6 show why time is the scarcest resource in the build.

What AI adds that the comparison does not price. The comparison holds scope and quality equal. In practice, AI also changes what a team can afford to do.

- **Faster experiments.** Prototypes cheap enough to build and throw away, so more ideas are tested before the contracts lock in the first three months and the feature set freezes at the beta.
- **More design options.** For each irreversible decision, AI drafts several alternatives and the scenarios that break them, so architects choose among options instead of defending the first one.
- **Scenarios and evaluations at scale.** Failure scenarios, permission matrices, partition tests and agent evaluation suites in numbers a team without AI would not have time to write. The testing plan in Part 3 depends on them.
- **Recommendations.** Test prioritization, cost and rightsizing analysis, and review of every change against the contracts, running continuously rather than waiting for a review meeting.
- **Faster adjustment.** When a design partner exposes a gap, the change, its tests and its documentation arrive in days, which keeps more of the beta period for proof.

A team without AI would most likely cut these to hold its schedule: fewer scenarios, fewer options considered and thinner evaluation of the agents. So the comparison understates what AI adds.

Two cautions keep it honest. The savings depend on the working rules in Part 3: contracts before code, human-owned tests and engineers who understand what they ship. DORA links AI adoption to higher throughput and to rising instability,⁶ and the METR result shows how easily perceived speed and real speed diverge. And AI does not compress the evidence calendar, which is why it saves about eight months on a program of 44 rather than half of it.

Claude's take: *Claude makes this build faster and cheaper, and the most valuable thing it gives is time. Launching eight months earlier is worth more than the \$3.7 million, because the market does not wait. What it cannot do is make the proof arrive sooner.*

Key takeaway: About \$17.9 million to reach launch on a smooth run, nearer \$26 million with a typical overrun, and about \$4.1 million a year after that. That is the entry ticket. People are 78% of the build. AI tokens are under 5%. The cost of writing code fell; the cost of a platform did not. Without AI, the same platform would launch about eight months later and cost about \$3.7 million more, and the later launch widens the gap the build has to close.

Part 5. The Market

In this part: *why parity is a moving target, why a build's specification has to freeze while the market keeps moving, what being behind has cost others, why keeping pace still leaves a build behind, and why agentic AI makes the leader's learning the advantage a newcomer cannot buy.*

The X factor: Chasing a moving target

This is the variable most build plans leave out. The platform described in the reference is not hypothetical. Buyers can purchase it today, and it keeps improving. Every estimate above measures the distance to where the market stands now. By the time we reach that line, it has moved.

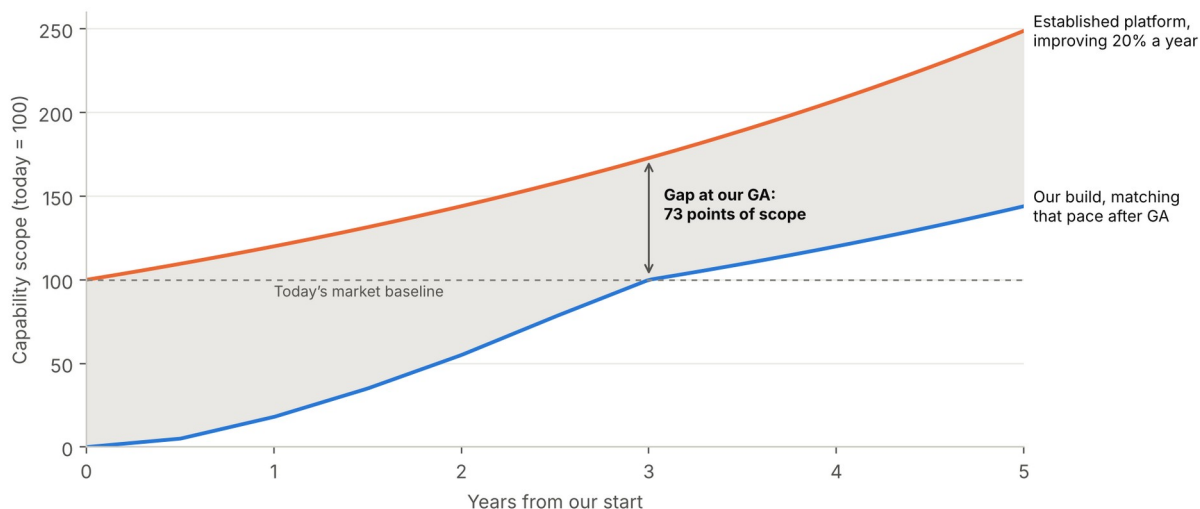


Figure 17. Illustrative only. If an established platform improves 20% a year, a build that reaches today's scope in three years arrives 73 points behind, and stays behind unless it moves faster from then on.

At 20% a year, parity at our general availability date would require delivering 73% more than the plan above, with the same team in the same time.

The specification freeze

Every build fixes its specification at some point. Ours starts from the periodic table as it stands today, and the contracts in Step 1 lock in the object model within the first three months. Scope can still grow for a while after that, at a cost to the schedule, but not indefinitely. Once the beta begins at month 20, the platform is in the hands of design partners and the team is hardening it: testing at scale, closing security findings and preparing for audit. Adding features at that stage resets the testing they depend on and puts the launch date at risk. In practice, the feature set freezes around the beta.

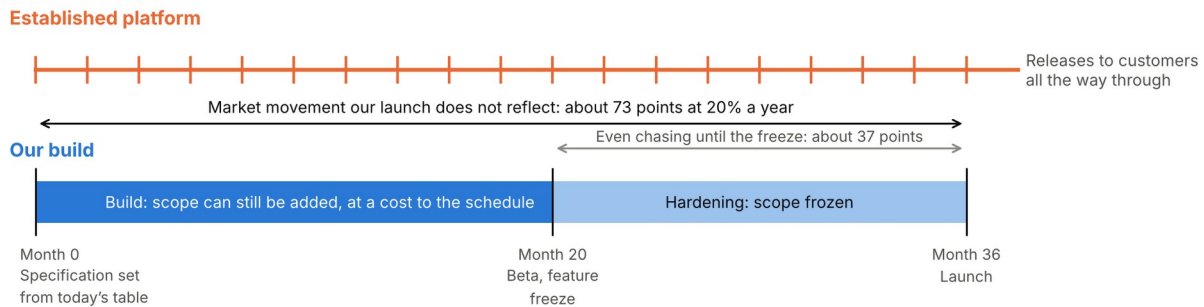


Figure 18. Illustrative. The established platform keeps releasing to customers throughout our build. Our specification is set at month 0 and our scope freezes at the beta, so whatever the market adds after that is missing on launch day.

The established platforms have no freeze. They release continuously to customers who already run them, and every release is tested against production use. Even a team disciplined enough to chase the market right up to its freeze would launch with a gap: at 20% a year, the 16 months between our beta and launch are worth about 37 points on their own. A team that holds to its original specification, as ours does, launches 73 points behind.

The pace is not standing still either. Established vendors now apply AI to their own roadmaps, and the rate of change in this category is rising. At 25% a year, the gap at launch grows to 95 points. At 30%, it reaches 120.

That gap decides the sale. Buyers comparing a new platform with one that already does more, and has customers to prove it, have little reason to choose the newcomer. Part 6 prices what closing the gap would cost.

What being behind has cost others

The pattern is not hypothetical. Two of the best-funded companies in technology have lived it.

Microsoft and Windows Phone. Windows Phone 7 reached the market in late 2010, about three years after the iPhone and two years after the first Android phone. It was well reviewed, and it arrived into an app gap: buyers chose phones for their apps, developers built for the platforms that had the buyers, and a late platform with few users could not break the loop.²⁴ Microsoft bought Nokia's phone business for \$7.2 billion to force its way in. In 2015 it wrote down \$7.6 billion of that purchase, took a restructuring charge of about \$800 million and cut 7,800 jobs, mostly in phones.²⁵ By the end of 2016, Gartner put Windows phones at 0.3% of smartphones sold.²⁴

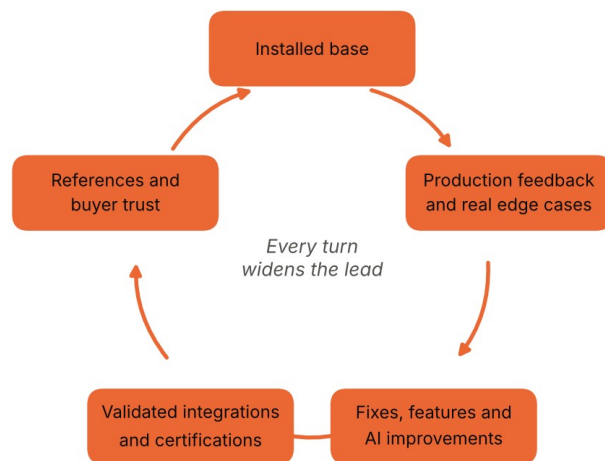
Google Cloud. Google entered cloud infrastructure years after Amazon Web Services, with some of the strongest infrastructure engineering in the world. Catching up still meant years of losses at scale. When Alphabet first disclosed the figures, Google Cloud had lost \$4.3 billion in 2018, \$4.6 billion in 2019 and \$5.6 billion in 2020.²⁶ It lost a further \$3.1 billion in 2021²⁷ and reported its first quarterly operating profit in 2023.²⁸ More than \$17 billion of operating losses in four years left it third in the market, behind Amazon and Microsoft.

Both companies had what a new control plane builder lacks: a vast installed base in other markets, deep capital and a brand buyers already trusted. Microsoft still could not close a three-year gap. Google closed part of its gap at a cost measured in billions of dollars and a decade of effort. The lesson for a build is not that catching up is impossible. It is that the market, not the build plan, sets the cost of catching up, and that cost is far larger than the build.

The periodic table is a snapshot of the market as it stands today. Every element in it is something buyers can already purchase. By the time our 104 elements are tested and launched, the platforms the table describes will have added depth to many of those elements and capabilities the table does not yet contain.

We build in a vacuum. For the first 20 months our platform has no users. Its requirements come from a specification, not from production. From month 20 to month 36 it has a handful of design partners. The established platforms spend the same three years in the market, and the market teaches them things a specification cannot.

The established platform: a compounding loop



Our build over the same period

Months 0 to 20

No users. Requirements come from the specification, not from production.

Months 20 to 36

A few design partners. Feedback from a handful of environments, not a market.

At GA

No installed base, no references, no partner certifications, no production track record.

Meanwhile

Buyers who standardize on another platform face switching costs, so the window narrows.

Figure 19. The established platform improves through a compounding loop. Each customer exposes edge cases, each fix and integration makes the platform more credible, and each reference brings more customers. Our build spends the same period outside that loop.

- **Learning.** Every production environment surfaces failure modes, provider quirks and workflows that no specification anticipates. An installed base turns those into fixes continuously. We discover them after launch, with our first customers paying for the lesson.
- **Enhancing.** Established teams use the same AI tools we do, applied to a working architecture and a backlog shaped by real customers. Their AI investment compounds on top of what already works.
- **Installed base.** Customers who standardize on a platform build processes, integrations and skills around it. Switching has a cost, so each one they win narrows the market we will be selling into at launch.
- **Ecosystem and trust.** Partner validations, certified integrations, analyst coverage and customer references take years to accumulate. At launch we have none of them, and enterprise buyers ask for all of them.

The natural response is that AI will let a new team move faster. It will. It will also let the established team move faster, and that team starts with more for AI to multiply: a working architecture, years of tests and failure cases, live customer feedback and engineers who already know which edge cases matter. DORA's amplifier finding cuts the same way. A multiplier applied to both teams does not change who is ahead. It changes how quickly the gap moves in the direction it is already moving.

A greenfield team can sprint early. It also inherits every Day 2 obligation for what it has already shipped, so each release adds maintenance load at the moment it is trying to accelerate.

The clock costs money twice. For every year of the build, the organization runs without the capability it is building: no governed self-service, no GPU reservations, no automated control over idle accelerators. Delivery is not adoption either. Spotify's head of engineering for Backstage has said the average Backstage adoption rate is stuck at 10%,²⁹ a reminder that building an internal platform and getting people to use it are separate problems. Gartner expects 80% of large software engineering organizations to have platform engineering teams by 2026, up from 45% in 2022.³⁰ Whether to have a platform is settled. Where engineering time goes is the open decision.

Keeping pace is not enough

Suppose the build launches and then improves exactly as fast as the established platform, 20% a year, with the same AI. It stays three years behind in time. In capability, the gap keeps growing, because improvement compounds: 73 points at launch, about 105 two years later and about 151 four years later. The ratio holds steady, with the leader doing about 1.7 times as much at every point, but the amount of capability the newcomer is missing keeps rising. Closing the gap means running faster than the leader, not as fast.

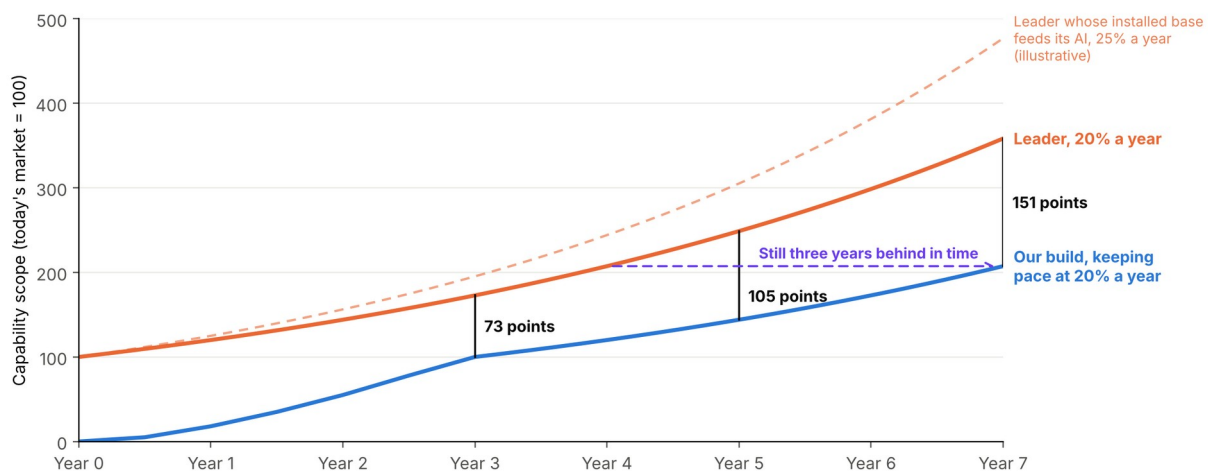


Figure 20. Illustrative. A build that matches the leader's pace after launch stays three years behind in time while the capability gap widens from 73 to 105 to 151 points. The dashed line shows a leader whose installed base lifts its pace to 25% a year.

Keeping pace is not cheap either. Keeping today's scope current, adding capability at 20% a year and running a go-to-market team brings the total to about \$44 million by year five and about \$68 million by year seven. At every point the build is still three years behind. Part 6 sets these costs beside the cost of drawing level.

The new advantage: what a platform has learned

For decades, catching up with a leader meant beating it on things people could see and build. A challenger studied the incumbent, then made something better, simpler, cheaper, more open or easier to integrate, offered better support or a better user experience, or changed the business model. Every one of those levers was a human, manual effort, and money could buy each of them. The cost model in this paper prices that race.

Agentic AI adds a race that money cannot buy into. It helps to separate two layers. The model, the large language model at the center, is fixed between releases, and every company can rent the same one. The agent system built around it is not fixed. It accumulates knowledge as it works, and it acts on what it has learned without waiting for a new model. In an infrastructure control plane, that learning takes four forms.

- **Environment knowledge.** Agents collect what is specific to each customer: its estates, its dependencies, its quirks and the way its teams actually work.
- **Applied judgment.** They fold that knowledge into their decisions, so the same request is handled differently, and better, in each environment.
- **Behavior and usage.** They track how infrastructure is used over time: where workloads go, when demand peaks and what sits idle.
- **The bigger picture.** They connect details that people manage in separate silos, such as how the network shapes data flows, how data flows load databases and how databases drive compute, and they weigh the impact of an action before taking it. Groups of agents working together speed this up.

Within each customer, that learning produces a platform tuned to how that organization works. Across customers, with security and data privacy controls, patterns from many environments improve the platform for all of them. The learning is private, because it comes from production environments no competitor can see. It is cumulative, because each day builds on the last. And it cannot be shortcut, because it comes only from running in the real world. This is the difference between an autonomous agent and a sophisticated workflow: a workflow does the same thing on day 300 as on day one, while an agent acts on everything it has learned since.

So a newcomer faces two gaps. The feature gap is the one this paper prices: engineers and time can close it, and money buys both. The learning gap cannot be priced. A rival that launches even six months after the leader is six months behind on learning on its first day, and the gap widens with every customer the leader serves. At each account, the incumbent's agents already know the environment, so a customer that switches gives up years of learned context.

Table 11. The two gaps a newcomer faces.

	The feature gap	The learning gap
What it is	Capabilities the leader has and the newcomer lacks	What the leader's agents have learned in production and the newcomer's have not
Where it comes from	Engineering	Customers running the platform every day
Can money close it	Yes: about \$55 million by year five, against a leader improving 20% a year	No. It can only be accumulated
How it changes	Grows in absolute terms unless the newcomer outruns the leader	Grows with every customer the leader adds

The learning gap also feeds the feature gap. A leader whose agents learn from a growing installed base improves faster. In our model, if that learning lifts the leader's pace from 20% to 25% a year, the gap at launch grows from 73 to 95 points, drawing level two years after launch costs about \$61 million and the engineering team needed grows from about 37 to about 46. At 30% a year, it is about 120 points, \$68 million and 57 engineers. Even then, the newcomer has bought the features, not the learning.

Two cautions keep this honest. Learning has to be governed: agents can learn the wrong lessons as easily as the right ones, so the advantage belongs to platforms that make their learning visible, evaluated and reversible. And leapfrogging still happens when a whole category changes, as software delivered as a service did to on-premises software. Within a category, a head start now tends to widen, and the platform that is learning is best placed to lead the next shift.

Claude's take: *Claude does not learn between conversations. Its model is fixed until a new one is released. The agent systems built on models like it are different: they keep what they learn and act on it. Every competitor can rent the same model. No competitor can rent what another platform's agents have learned.*

Key takeaway: Our specification freezes while the market keeps moving, so we launch three years behind: 73 points against a platform improving 20% a year. Keeping pace from there keeps us three years behind. Behind the feature gap sits a learning gap: established platforms with agentic AI learn from every customer, every day, and that learning cannot be bought or copied. Microsoft and Google show how expensive a feature gap is to close. A learning gap is harder still.

Part 6. What We Found

In this part: *the outcome measured against our three conditions, the numbers in one place, the real cost of catching the market, the price of staying behind, the risks of chasing a leader, what we would and would not achieve, and what this means for the belief that AI will do all of this anyway.*

The outcome

Claude and I could plan this platform, and with the right team we could build it. Measured against the periodic table, we would deliver all 104 of today's elements, tested and production-ready, at launch. It takes three years, 23 people at peak, about \$17.9 million on a smooth run, nearer \$26 million with a typical overrun, and about \$4.1 million a year to keep it current. A competitive one, on the day it launches, is a different matter. The build launches three years behind. Keeping pace from there costs about \$68 million by year seven and leaves it three years behind. Drawing level on features costs about \$55 million by year five, and no budget closes the learning gap.

Table 12. The outcome against our three conditions for done.

Condition	Verdict	Why
Production-ready	Achievable by month 36	With contracts first, a governed execution engine, tested failure behavior, independent security review and a team on call
Fully tested	Achievable, on its own calendar	Continuous tests start early, but real-world proof runs from the beta at month 20 to launch at month 36 and cannot be compressed by writing code faster
Competitive	Not achievable at launch	We would match today's table, which describes the market as it stood when we started. The established platforms would have spent the same three years learning from customers, improving and growing their installed base. Catching them costs about three times the build

The numbers in one place

Table 13. The numbers in one place.

Measure	Our estimate
Time to general availability	36 months, with an internal alpha at month 12 and a design-partner beta at month 20
Team	23 people at peak, about 56 person-years
Total cost to general availability	About \$17.9 million on a smooth run, with a range of \$13 million to \$25 million
With a typical overrun (45% over budget, 7% over time)	About \$26 million, with a range of \$19 million to \$37 million, and launch at about month 39
People	About \$13.9 million, 78% of the total
AI usage (tokens for coding, review and agent testing)	About \$0.8 million, under 5% of the total
Cloud and GPU environments, lab, tooling, recruiting, security and audit	About \$3.1 million
Cost to run and improve the platform after launch	About \$4.1 million a year
An AI-built demo of the same idea	Less than 2% of the total
The same build without AI (mid case)	About 44 months and \$21.6 million: launch 8 months later and \$3.7 million more
Elements of the table delivered and tested	32 at the internal alpha, 54 at the beta, all 104 at launch
Average cost per element of the table	About \$172,000

Measure	Our estimate
Gap at launch against a platform improving 20% a year	73 points of capability scope
Cost to compete with that platform two years after launch (illustrative)	About \$55 million in total, 3.1 times the build, including the build itself
Total spent by year five, keeping pace with that platform	About \$44 million, and still three years behind
Total spent by year seven, keeping pace with that platform	About \$68 million , and still three years behind
The learning gap	Not priced. Three years of the leader's production learning at launch, growing with every customer

Planning estimates for this hypothetical build, by the authors. They are not measured results and do not describe any vendor's investment or value, including Quali's.

The real cost of catching up

Our plan answers one question precisely: what it costs to build what the market offers today. The answer, about \$17.9 million over three years, buys parity with a snapshot. It gets us to where the market stands now, three years late. It does not buy the position the established platforms will hold on the day we launch, and that is the position buyers will compare us against.

Part 5 showed why the gap opens and what it has cost others. To price it for our build, Claude and I extended the model past launch. It is illustrative, not a forecast, and platforms do not improve in straight lines. Leaving the chase out of the estimate would understate the decision far more than any assumption inside it.

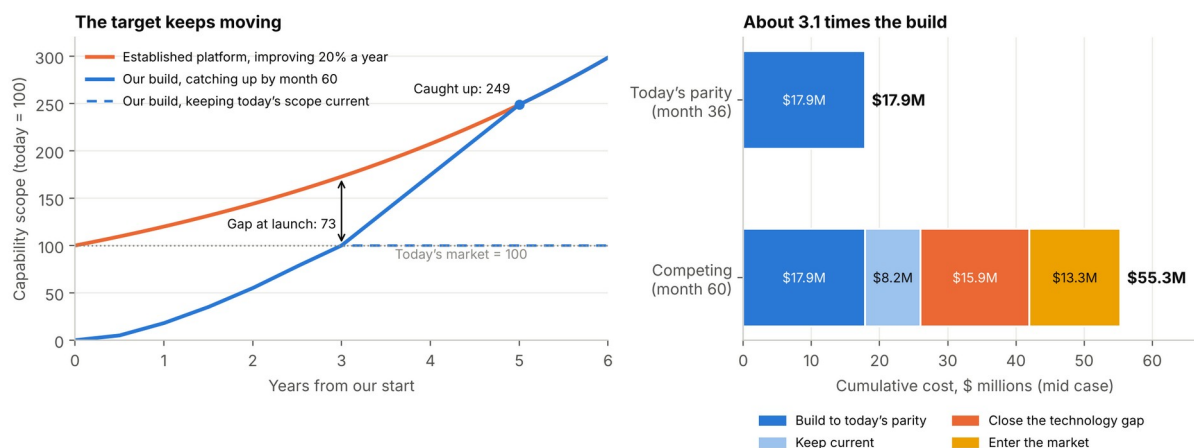


Figure 21. Illustrative, mid case. Left: against a platform improving 20% a year, our build launches 73 points behind and must deliver about 149 points in two years to draw level at month 60. Keeping today's scope current never closes the gap. Right: what drawing level costs, against the cost of reaching today's parity.

The extension adds three costs to the build.

- **Keeping current.** About \$4.1 million a year to run the platform and keep today's 104 elements working as providers, hardware and threats change. This buys no new ground.
- **Closing the technology gap.** Everything the established platform adds while we build and while we chase, delivered on top of what we built. We priced each point of capability at the build's own average cost, then took 40% off, because the contracts, engine, governance and test harnesses already exist. At 20% a year, drawing level two years after launch means delivering about 149 points

in 24 months: half again the original scope, in two-thirds of the time, with an engineering team of about 37, 60% larger than the build's peak.

- **Entering the market.** A product competes only when buyers consider it, and buyers decide early. 6sense's 2025 study of nearly 4,000 B2B buyers found the eventual winner was already on the buyer's first shortlist 95% of the time, and buying cycles averaged 10.1 months.³¹ A new platform with no installed base, no references and no analyst coverage starts outside those shortlists. We costed a 15-person go-to-market team, covering sales, solutions engineering, marketing, customer success and partner alliances, at half strength from the beta and full strength from launch, plus \$1 million a year for analyst relations, events, certifications and content.

Table 14. What it costs to compete at month 60.

To compete at month 60, against a platform improving 20% a year	Mid case
Build to today's parity, months 0 to 36	\$17.9 million
Keep today's scope current, months 36 to 60	\$8.2 million
Close the technology gap, months 36 to 60	\$15.9 million
Enter the market, months 20 to 60	\$13.3 million
Total to compete	About \$55 million, 3.1 times the build

Illustrative extension of our planning estimates. Capability is measured on a scale where today's market equals 100. Engineering and go-to-market roles are costed at the same fully loaded rate. Installed base, brand and partner ecosystem are not priced, because money alone does not produce them.

Three findings come out of the extension.

- **The build estimate is the smallest number in the decision.** Reaching today's parity is about \$17.9 million. Competing with the market as it will be costs about \$55 million at the default settings, and between about \$35 million and \$113 million for platforms improving 10% to 30% a year, drawing level one to four years after launch.
- **Speed and cost pull against each other.** Drawing level one year after launch needs an engineering team of about 48, twice the build's peak, at the point where the platform is newest and its Day 2 load is rising. Waiting longer lowers the team size but raises the total, because the target keeps moving: each extra year adds \$13 million to \$15 million.
- **Money does not buy an installed base.** A funded go-to-market team still waits through buying cycles, earns references one customer at a time and sells against platforms whose customers have already built processes, integrations and skills around them. No one buys a platform that launches three years behind with no customers, and that is the starting position.

For a team building for its own organization, the market gap takes the form of adoption. The platform still has to win its engineers away from scripts, tickets and the tools they already know, and the Backstage figure in Part 5 shows how hard that is.

Claude's take: *The build estimate buys the starting line the market crossed three years earlier. Catching the field costs several times more, and the part money cannot buy, an installed base that trusts the platform, takes the longest.*

The price of staying behind

Part 5 showed that keeping pace leaves a build three years behind. Put the money and the time together and the choice looks like this. Totals after launch include keeping today's scope current, new capability at the leader's pace and the go-to-market team described above.

Table 15. What it costs to stay behind, and what it costs to draw level.

When	Total spent (mid case)	Where the build stands
Year 3, launch	\$17.9 million on a smooth run, about \$26 million with a typical overrun	Three years behind: 73 points
Year 5, keeping pace and selling	About \$44 million	Still three years behind: 105 points
Year 7, keeping pace and selling	About \$68 million	Still three years behind: 151 points
Year 5, outrunning the leader	About \$55 million	Level on features only, with no customers and no production learning

And that is before the rest of the business. These totals cover building the product, keeping it current and a 15-person go-to-market team. Turning that into a business people buy from also needs a full sales force and channel, marketing at a scale that wins shortlists, customer support, onboarding and professional services, certifications and compliance in every market served, the corporate functions behind them and years of selling before revenue covers the cost. Each depends on the business being built, so we do not estimate them.

The risks of chasing a leader

Part 3 set out the ten biggest risks inside the build. Entering a market where a leader is already running adds risks the build plan does not cover, and the newcomer carries all of them at once.

- **Delivery.** The build slips, overruns and depends on scarce engineers, and early design mistakes in the data model, tenancy or state are expensive to undo once customers depend on them. AI adds its own: generated tests that confirm generated code, and speed that feels faster than it is.
- **Market.** The leader keeps shipping while the specification is frozen, and it can respond with price, bundling or features aimed at the newcomer. Buyers shortlist the platforms they already know, and buying cycles push revenue well past launch.
- **Financial.** The capital required keeps rising while the platform is still behind. Every year of delay adds cost and widens the gap at the same time, and the money and people could have gone into the core business.
- **Strategic.** After \$20 million or more, stopping feels like failure, so programs run past the point where they should stop. The likely end state is a platform maintained for internal use rather than one that leads, and senior attention goes to catching up instead of to what differentiates the business.

The risks are lopsided. The leader has already absorbed its delivery risk and proved its market. For the newcomer, the four reinforce each other: a delay is a cost overrun and a wider gap at the same time.

What we would achieve

- **A working, governed platform.** Self-service infrastructure across cloud, data center and GPU estates, with identity, policy, cost and audit built in from the first request.
- **Tested failure behavior.** Partial failures, retries, concurrency and network partitions handled by design and proven by test, not discovered by customers.
- **A governed agentic layer.** Agents that act through the same identity, policy and audit as people, with evaluated limits on their autonomy.
- **Independent evidence.** Penetration tests, a SOC 2 Type II report and a record of design-partner use.

- **A team that understands the system.** Engineers who can diagnose and change the platform safely, whoever wrote each line.
- **Faster progress than before AI.** AI would shorten the time from decision to working code at almost every step, most of all in the middle of the build. Against the same plan without AI, it brings launch about eight months earlier and saves about \$3.7 million.

What we would not achieve

- **Parity on launch day.** We would arrive three years behind, in a market that had moved on. Keeping pace would keep us there, and drawing level would cost about three times the build.
- **An installed base or a track record.** No references, no partner certifications and no production history at scale. Enterprise buyers ask for all three.
- **Production learning.** The leader's agents learn from every customer environment, every day. That learning is private and cumulative, no budget buys it, and ours would start only with our first customers.
- **A shorter evidence calendar.** Audits, soak tests and real-world validation run on their own clocks.
- **A cheap platform.** AI lowered the cost of code. People, infrastructure and assurance still account for more than 95% of the bill.
- **An end date.** The platform keeps costing about \$4.1 million a year, because providers, hardware and customer needs keep changing.

Why "AI will do all this anyway" is only partly true

The belief we set out to test contains real truth, and it deserves a fair hearing. AI does write most of the code. Prototypes are now cheap and fast. Several steps in our plan move markedly faster because of it. Where the belief breaks down is in assuming that writing code is most of the work.

Table 16. Where “AI will do all this anyway” holds, and where it does not.

The assumption	What is true	What is not
AI will write the platform	AI can draft most of the code and tests	Code is the step with the most AI leverage, but deciding, proving and owning the result remain with people at every step
AI makes it cheap	Token costs are modest, under 5% of our build	People, infrastructure and assurance are the other 95%, and they do not fall at the same rate
AI makes it fast	Time from a clear decision to working code shrinks	Real-world testing, audits and design-partner validation run on fixed calendars
AI can test it	AI generates tests and failure scenarios at scale	Proof requires real cloud accounts, real hardware and independent reviewers, and generated tests tend to confirm generated code
AI can run the agents safely	AI builds agents and evaluation harnesses quickly	People must set the limits of autonomy and prove agents stay inside them under adversarial input
AI will close the gap with established platforms	AI accelerates a new team	It accelerates established teams too, and they start with more for AI to multiply: architecture, customers, production history and trust
Our agents will learn as fast as theirs	Once running, a new platform's agents can learn as quickly as anyone's	They learn from production. The leader has years of customers feeding that learning, and the newcomer starts from zero
AI will keep getting better	Model capability is improving quickly, and some of our people estimates will fall	The structural constraints do not depend on model capability: real systems, fixed calendars, accountability and market position

Claude's take: *Claude would want to be on this team, and would make it faster at almost every step. But the honest answer to whether AI can build this is that AI can write much of it. Building it means deciding what it should be, proving it works on real infrastructure, standing behind it in production and earning the trust of the people who run on it. Those parts still belong to people, and they take time no model can remove.*

Where Torque fits

Quali developed Torque, so I have a point of view. Torque was built by software engineers with long, deep experience in infrastructure platform development, and that work is now accelerated with Claude and other AI tools. It did not assemble itself. It came from sustained collaboration between people and AI across design, implementation, testing and refinement, against real infrastructure and real customer environments.

That collaboration continues, aimed at the hardest steps in this plan: using AI to build the platform, governing the AI agents that act on infrastructure, and delivering AI infrastructure as a governed service.

None of this means nobody should build. A focused service that encodes something unique to your business can be the right call. The test is simple: build where you differentiate, and adopt where someone else is already running ahead.

The last word

AI has changed the cost of writing code. It has not changed the cost of proving that infrastructure software is safe to trust, or the distance the market travels while you prove it. Building a control plane

with AI is possible. Building one that is competitive on the day it launches means catching a target that AI is helping to move, and the \$17.9 million that reaches today's market is the first third of that bill. Agentic AI adds a gap no bill covers: what the leader's platform learns from its customers every day. You can copy a platform. You cannot copy what it has learned.

Whichever path you choose, start with the [interactive Infrastructure Control Plane reference](#). It contains no tracking or activity logging but what it does is show exactly what you are signing up for.

Sources

1. [Kubernetes documentation, Controllers](#)
2. [Temporal documentation, Activity Definition \(idempotency\)](#)
3. [IntuitionLabs, H100 rental prices compared across cloud providers](#)
4. [Gartner, Over 40% of Agentic AI Projects Will Be Canceled by End of 2027](#)
5. [Vanta, How long does a SOC 2 audit take](#)
6. [Google Cloud, 2025 State of AI-assisted Software Development \(DORA\)](#)
7. [Stack Overflow, 2025 Developer Survey: AI](#)
8. [U.S. Bureau of Labor Statistics, Occupational Outlook Handbook: Software Developers](#)
9. [U.S. Bureau of Labor Statistics, Employer Costs for Employee Compensation, June 2026](#)
10. [Anthropic, Claude Code: Manage costs effectively](#)
11. [Anthropic, Claude API pricing](#)
12. [Amazon Web Services, Amazon EKS pricing](#)
13. [DoiT, AWS EC2 m7i.xlarge pricing in us-east-1](#)
14. [GitHub, Actions runner pricing](#)
15. [Datadog, Pricing](#)
16. [Spheron, AMD MI300X and MI355X pricing](#)
17. [Blaze Information Security, Penetration testing cost and pricing](#)
18. [Scrut, How much does SOC 2 compliance cost](#)
19. [McKinsey & Company with the University of Oxford, Delivering large-scale IT projects on time, on budget, and on value](#)
20. [Paradis et al., How much does AI impact development speed? An enterprise-based randomized controlled trial \(Google\)](#)
21. [Cui, Demirer, Jaffe, Musolff, Peng and Salz, The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers](#)
22. [McKinsey & Company, Unleashing developer productivity with generative AI](#)
23. [METR, Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity](#)
24. [ADTmag, Windows Mobile market share and the app gap](#)
25. [TechCrunch, Microsoft writes down \\$7.6B of its Nokia acquisition, announces 7,800 layoffs](#)
26. [TechCrunch, Google Cloud lost \\$5.6B in 2020](#)
27. [ITPro, Google Cloud lost \\$3.1 billion in 2021](#)
28. [CNBC, Google's cloud business turns profitable for the first time on record](#)
29. [The New Stack, Spotify's Backstage Roadmap Aims to Speed Up Adoption](#)
30. [Gartner, What Is Platform Engineering](#)
31. [6sense, 2025 B2B Buyer Experience Report](#)