Solo-Building a 40-Million-Row AI Operating System: Architecture, Scale, and the Discipline That Made It Maintainable

Abstract

We report the architecture, scale, and engineering discipline of ARQERA, a governance-first AI-operations platform built and operated primarily by one engineer working with AI agents. The system is large by any measure: **2,688 API endpoints across 402 routers, 805 service modules, 170 data models, ~653,700 lines of backend Python and 1,440 frontend modules**, backed by a **~40-million-row evidence substrate**. It is deployed GitHub-independently through a bespoke pipeline with **seven fail-closed gates**, across a multi-cloud footprint (managed Kubernetes, a serverless inference plane, and a self-hosted homelab of GPU nodes running the platform's own CI runners). The central engineering finding is about *sustainability at this scale with this team size*: the platform's fix-to-feature commit ratio, historically **2.1 : 1** (an early warning that maintenance was consuming the effort), was driven down to **0.88 : 1** across 4,742 commits — more features than fixes — by codifying a small set of pre-commit disciplines and *encoding them as automated deploy gates* rather than leaving them to attention. We describe the gates, the deploy mechanics that made rapid iteration safe (16 backend deploys in a single session), and the performance-engineering pattern (route reads through maintained projections, never the raw ledger) that turned a class of production timeouts into sub-second responses (up to 300×). The report's thesis: **at solo scale, the binding constraint is not building the system but keeping it maintainable, and the leverage is in converting discipline from a practice into a gate.**

1. What was built

ARQERA is an AI-native operating system: users direct a workforce of governed AI agents to do real operational work, under an evidence-emitting governance layer, on their own model keys. The measured surface:

Dimension	Count
API endpoints	2,688
API routers	402
Service modules	805
Data models	170
Celery (async) tasks	180
Backend Python	~653,700 LOC
Frontend TypeScript modules	1,440
Evidence substrate (top tables)	~40 M rows / ~40 GB
Distinct evidence artifact types	252

This is not a prototype's surface area; it is a mid-size company's. The engineering question this report addresses is not "can one person build this" — evidently, with AI agents, one person did — but **"can one person keep this maintainable,"** which is the harder and more interesting problem.

---

2. The maintainability problem, quantified

Every large codebase fights entropy; a solo-plus-agents codebase fights it with a fraction of the usual review capacity. The measurable symptom is the **fix-to-feature ratio** — commits prefixed `fix` versus `feat`. A ratio above 1 means the team spends more effort repairing than building; a ratio well above 1 is a codebase eating itself.

ARQERA's history shows both the danger and the recovery. An early window carried a fix-to-feat ratio of **2.1 : 1** — for every feature, more than two fixes — an explicit signal, recorded at the time, that rework was dominating. Across the full history of **4,742 commits**, the ratio is now **0.88 : 1** (1,842 `feat`, 1,616 `fix` in the deployment repository; 0.85 : 1 in the application repository) — *more features than fixes*. Something drove the ratio down by more than half. That something was not more discipline in the abstract; it was **discipline encoded as automation**.

---

3. The gates: discipline as code

The core move was to take the pre-commit disciplines that a careful engineer *should* follow — and reliably will not, at 2,688-endpoint scale, on limited sleep — and make them *mechanical gates a deploy cannot pass*. The deployment pipeline runs **seven fail-closed gates**, each born from a specific class of production incident:

1. **Requirements-drift gate.** The build layers application code onto a pre-built base image and never runs `pip install`; a changed dependency would silently be absent in production. The gate refuses to deploy unless

the base's recorded dependency hash matches the current requirements — turning a silent missing-dependency `ImportError` into a named, blocking failure.

2. **Layer-depth guard.** The copy-on-base build adds three image layers per deploy against a hard 127-layer ceiling; the gate refuses a deploy that would breach it (a real incident: a silent build failure at the limit) and prints the flatten procedure.
3. **Smoke test.** The image is booted in a throwaway container and asserted to construct the application and load all routers before it is allowed near the cluster.
4. **Health gate.** Post-rollout, the live pods must answer `/health` `200`, targeting the correct container (a sidecar at index 0 previously produced false failures).
5. **Schema-drift gate.** The single most valuable gate. It was born from a production login `500` — a column the code read did not exist in the live database because the code shipped ahead of its migration. The gate compares every model column the new image reads against the live schema and **refuses to declare a deploy done if the code is ahead of the schema.** It converts a silent production `500` into a blocking, named deploy failure.
6. **Plan-catalog gate.** Born from a checkout `500` (a stale, hand-set payment-price identifier that orphaned checkout). The gate reconciles the canonical pricing definition against the live payment provider and fails closed if any purchasable tier cannot resolve to a live price.
7. **Version + provenance stamp.** The deploy stamps the build tag, git SHA, and build time into the running process so the system can self-report what is live (the subject of the companion wieldability paper) — closing the loop that made "what version is running?" require container archaeology.

Each gate is a lesson made permanent. The pattern is general and, we argue, is *the* leverage for maintainable solo-plus-agent engineering: **do not rely on an engineer (or an agent) remembering the discipline; encode the discipline where forgetting it is impossible.** The fix-to-feat recovery from 2.1 : 1 to 0.88 : 1 is the quantified return on that pattern.

---

4. Deploy mechanics: making fast iteration safe

Gates only help if deploys are cheap enough to run often. Two mechanics made a 16-deploy session (backend versions `m147` through `m162` in a single working session) safe rather than reckless:

- **GitHub-independent, copy-on-base builds.** Deploys assemble code onto a pre-built dependency base, push through a governed registry door, and roll via a governed Kubernetes door — no dependency on external CI availability, and a fast build because dependencies are inherited, not reinstalled.
- **Single-revision rollouts.** A subtle but consequential fix: stamping the version environment variable and setting the image as *two* separate operations produced *two* rollouts (a double roll that blew a timeout). Combining image and environment into a *single* strategic-merge patch yields one rollout. This is a microcosm of the whole report — a small, boring correctness detail (one revision, not two) is the difference between a deploy that lands cleanly and one that appears to hang.

Critically, every backend deploy rolls the API *and* the async workers *and* the scheduler together to the same version — a discipline born from a specific drift incident (the API on a recent version while background workers ran code more than a hundred releases stale, silently executing outdated logic). At solo scale, *partial* deploys are how a system quietly desynchronises from itself.

5. Performance engineering: read projections, not the ledger

A recurring production failure was pages timing out (rendered to users as "offline"). The root, across a family of endpoints, was the same: **reading the ~40-million-row substrate directly on the request path.** The fix pattern was uniform and is the report's reusable performance lesson:

> Never scan the ledger to answer a request. Read a maintained aggregate *projection*, an index-backed `EXISTS`/cutoff, or a planner row-count *estimate* — never the raw table.

Measured results from applying it:

Operation	Before	After	Factor
Chain verification (platform tenant)	45 s	2.3 s	≈19×
Recent-N via `OFFSET` → index cutoff	16.3 s	0.10 s	163×
Public endpoints (unauth)	60 s+	0.1–0.2 s	≈300×
Tenant `COUNT` → projection	44.75 s	7 ms	≈6,000×

Seventeen request-path scans were found and routed to projections or estimates; the worst were *public, unauthenticated* endpoints running 44-second aggregates on every anonymous hit — simultaneously a performance defect and a denial-of-service surface. The engineering discipline that surfaced these was not profiling in the abstract but *operating the system as a user and tracing every timeout to ground truth in the live database* — which also, once, caught a false measurement (a row-level-security-masked query reporting a table "empty" that in fact held 156 rows), reinforcing a meta-discipline: **verify the measurement harness before trusting the measurement.**

---

6. The multi-cloud footprint

The system spans three trust domains by deliberate tiering rather than uniform redundancy:

- **Managed Kubernetes** (a major cloud) for the stateful, HA-critical plane — the transactional database with point-in-time recovery, the application workloads. The principle: lean on managed-provider HA rather than self-triplicating.
- **A serverless inference plane** hosting the platform's own fine-tuned models (a small family of task-specialised 8B models behind an OpenAI-compatible endpoint) as one tier of a five-tier model-routing ladder.
- **A self-hosted homelab** — GPU workstation-class nodes — running the platform's *own* CI runners and burst training, treated as rebuild-if-lost convenience compute, not a single point of permanent loss.

The governing rule is a tiered recoverability doctrine: irreplaceable assets (signing keys, the evidence ledger, code) must have two copies in two trust domains with *tested* recovery; operational assets lean on managed-cloud HA; convenience assets keep one copy and are rebuilt if lost. The evidence ledger, the platform's core value, is

accordingly three-store replicated. The lesson for small teams: **redundancy is not free and should be spent where loss is permanent, not sprayed uniformly.**

---

7. Discussion

The interesting claim of this report is not the scale (AI agents make large surface area achievable by small teams) but the *maintainability* result: the same small team can keep a large system healthy **if discipline is converted from a practice into a gate.** Every one of the seven gates (§3) replaced a thing a human was supposed to remember with a thing a machine enforces; the fix-to-feat ratio halving is the measured consequence. This reframes what "engineering seniority" means in an agent-assisted world: less about writing more code (agents do that) and more about *designing the gates, projections, and single-source-of-truth structures that keep agent-written code from eating the system.* The highest-leverage artifacts in this codebase are not features; they are the schema-drift gate, the canonical-pricing source, the projection-read pattern, and the single-revision rollout — each a small structure that makes a whole class of failure impossible.

Limitations

This is an engineering report of one system by its builder, not a controlled study; the causal claim that the *gates* drove the fix-to-feat ratio down (versus other factors — the codebase maturing, the author learning, the commit-message conventions shifting) is argued from the timing and the incident-to-gate mapping, not isolated experimentally. "Fix-to-feat ratio" is a proxy for maintainability that is sensitive to commit-message discipline and can be gamed. The LOC and endpoint counts measure *surface*, not *quality* or *usage* — much of the 2,688-endpoint surface may be lightly used, and a smaller, better-factored system could be preferable. The multi-cloud tiering is described as doctrine; whether it is *lived* everywhere (versus aspired to) is exactly the kind of gap the platform's own audits repeatedly found. Finally, "solo" understates the AI agents' contribution: this is a report on solo-*plus-agents* engineering, a genuinely new mode whose norms — including how much of the author's skill is in *directing* agents versus writing code — this report describes more than it measures.

---

8. Conclusion

One engineer, working with AI agents, built and operates a 2,688-endpoint, 40-million-row AI operating system. The scale is a product of the tools; the *maintainability* is a product of discipline — specifically, of converting discipline into automation: seven fail-closed deploy gates, a projection-read performance pattern, single-source-of-truth structures for pricing and plans, and single-revision rollouts. The measurable return was a fix-to-feature ratio driven from 2.1 : 1 to 0.88 : 1 across 4,742 commits — a system building faster than it repairs. The transferable lesson for the agent-assisted era: agents let a small team *build* large; gates let a small team *keep* it. Design the gates.

Appendix A — Headline figures

Figure	Value
API endpoints / routers	2,688 / 402
Service modules / data models / async tasks	805 / 170 / 180
Backend Python / frontend modules	~653,700 LOC / 1,440 files
Evidence substrate	~40 M rows / ~40 GB / 252 types
Total commits (deployment repo)	4,742
Fix-to-feature ratio	**0.88 : 1** (from a historical 2.1 : 1)
Deploy gates (fail-closed)	7
Deploys in one session	16 (`m147` – `m162`)
Best performance improvement	`COUNT` 44.75 s → 7 ms (≈6,000×)
Trust domains (tiered, not uniform)	3

All figures measured live on the production platform and its repositories, 23 July 2026.