OPEN PLATFORM — WHITE PAPER — V1.0 — MAY 2026

Everything but the agent

A self-hosted developer platform for the era of agent-driven software

BY THE ESPO OPEN PLATFORM TEAM

ESPODEV.COM/WHITEPAPER

ESPO DEV

The race is to build the smartest agent. The harder, less glamorous race is to build the platform that lets one ship anything an ordinary person would call software — identity, source control, CI, deploys, ingress, databases, audit. The substrate. This paper describes one specific answer: a self-hosted developer platform on which more than a hundred real applications, written by humans and agents working side by side, have been built from a single shared template, behind a single identity provider, with every artifact traceable to the natural person who signed for it.

The argument in one page

The race is to build the smartest agent. The harder, less glamorous race is to build the platform that lets one ship anything an ordinary person would call software. Identity, source control, CI, deploys, ingress, databases, audit — the substrate. Most teams are renting that substrate from somebody else, which is fine until the day they need to prove who did what, or move, or just see inside. The platform described here is the other answer: one Forgejo for identity, one Flux for deploys, one Traefik for traffic, one namespace per app, all on a cluster you own. From there an agent — bring your favorite — can take a sentence of intent and turn it into a Forgejo repo, a green CI run, and a URL serving real code in roughly two minutes. The agent does the typing. The platform does the remembering. The accountability never leaves a real person.

The rest of this paper unpacks that paragraph. The architecture is small. The opinions inside it are not.

Why the AI race won't ship software by itself

The most expensive part of building a piece of software has moved. It used to live in the code itself — the design of a data model, the careful sequencing of state changes, the prose of a function name. Agents are very good at all of that. Cursor will draft a schema in twenty seconds. Claude Code will refactor a thousand-line module while you walk to the kitchen. The supply of code, for the first time in the history of the field, is no longer the bottleneck.

What is the bottleneck is everything around the code. Where does the source live? Who can sign in to read it? Where do the builds run? How does the running app authenticate users, talk to a database, store an uploaded file, get a TLS certificate, expose itself to the world on a real hostname? Who can prove, two years later, that a particular line was written by a particular person on a particular afternoon? On most teams the answer to those questions is a Lego pile of vendor accounts: GitHub for source, Vercel for hosting, Auth0 for login, AWS for the database, Cloudflare for the front door, and a different login URL for each. The Lego pile works fine for the first app and gets expensive — in money, in cognitive load, in audit gymnastics — for the hundredth.

> *The supply of code is no longer the bottleneck. The supply of places that code can safely land is.*

This paper is about the other place. A platform whose explicit job is to be the substrate underneath agents, not a competitor to them. The platform itself doesn't write code. It hosts code, runs code, identifies the human responsible for each piece of code, and remembers what happened. Everything else — the actual writing — is delegated to whichever agent the user prefers, or to no agent at all. The site you are reading this paper on (`www.espodev.com`) is itself an app on the platform; so is the portal where briefs are submitted; so are the more than a hundred other apps quietly running on the same cluster as you read.

Mental model: the four "ones"

The platform's design is small enough to fit on an index card. Four things, each of which is *exactly one*:

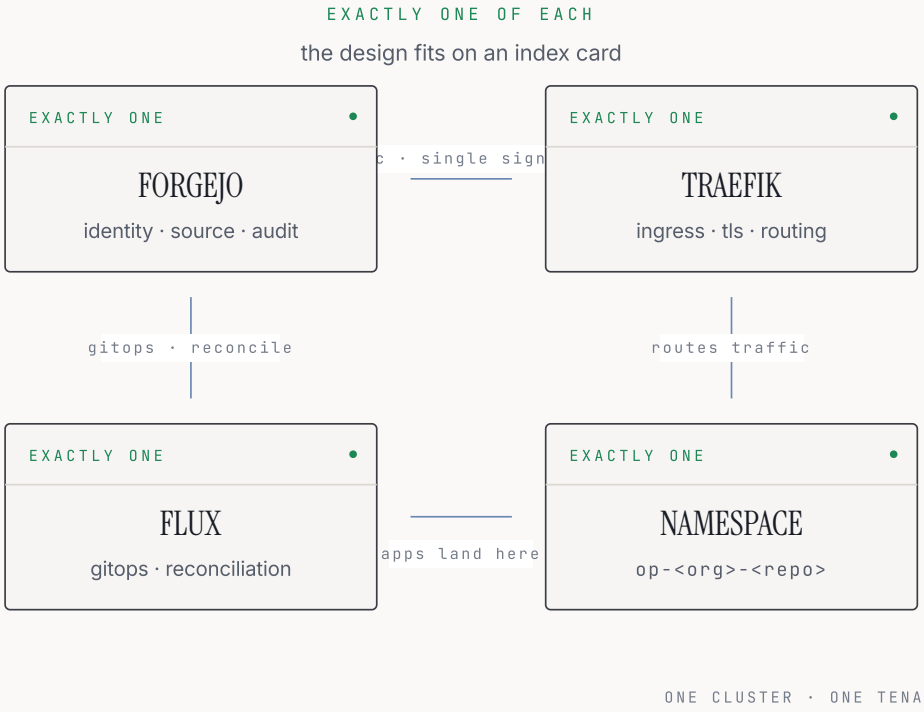

FIGURE 1 — THE FOUR 'ONES'. FORGEJO IS THE SINGLE IDENTITY ROOT. FLUX IS THE SINGLE DEPLOYMENT CONTROLLER. TRAEFIK IS THE SINGLE INGRESS. EACH APP GETS EXACTLY ONE KUBERNETES NAMESPACE, NAMED BY THE CONVENTION OP-<ORG>-<REPO>.

One identity provider. Forgejo, self-hosted, at `https://forgejo.{domain}`. Every app on the platform — the portal, the marketing site, the hundred-plus workload apps — authenticates against the same Forgejo via OAuth2/OIDC. There is no separate IdP. There are no fan-out identity adapters. A Forgejo personal access token is, at once, the user's Git credential, container-registry credential, and platform API bearer token. One human, one token, one identity.

One GitOps controller. Flux watches a Git repository called `system/open-platform` and reconciles the cluster against it on a roughly two-minute cycle. The contents of that repository *is* the cluster: every app's deployment, every preview, every secret reference, every ingress

rule. Nothing is applied to the cluster by hand. The platform docs put this bluntly: *"Never `kubectl apply` platform resources by hand. Flux will fight you and win."*

One ingress. Traefik terminates TLS for the platform's wildcard certificate and routes every request to the right service inside the cluster. Public traffic enters at one place. Internal service-to-service traffic uses Kubernetes DNS and never hairpins through public hostnames. There is no second ingress for "the API," no special path for "the marketing site." There is one front door.

One pattern for apps. Each app lives in exactly one Kubernetes namespace, named `op-<org>-<repo>` (and `op-<org>-<repo>-pr-<N>` for previews). The convention is load-bearing rather than cosmetic — platform tooling derives the namespace from the `(org, repo, pr)` triple without doing any lookup. Every app's namespace contains exactly one Deployment named `app`, one Service, one Ingress, and three Secrets called `app-db`, `app-s3`, and `app-oauth`. A platform engineer who has read one app's namespace has, by construction, read all of them.

These four choices generate a surprising amount of the platform's behavior. Auth is simple because identity has one source. Deploys are auditable because the cluster is a function of a Git repository. Ingress is uniform because there is only one of it. Operating tooling — "show me this app's logs," "is this app healthy," "what was the last successful CI run" — is a one-liner because the namespace pattern lets the tool address any app on the platform without configuration.

The natural-person tether

The defining design choice of the platform is not architectural. It is who is responsible for what.

Every call to the platform's MCP server carries `Authorization: Bearer <forgejo-pat>`. The token is not a service principal; it belongs to a specific human, issued from that human's Forgejo settings page. From that token the platform derives the actor for every downstream action: the commit that lands in the new repo, the CI run that builds it, the OCI image pushed to the in-cluster registry, the GitOps write that registers it with Flux, the deploy that puts it behind Traefik. There is no impersonation layer, no "agent identity" floating free of a person. The audit trail is not an additional feature; it is the only mechanism by which the work can happen at all. An agent that wants to ship code first needs a human willing to sign for it.

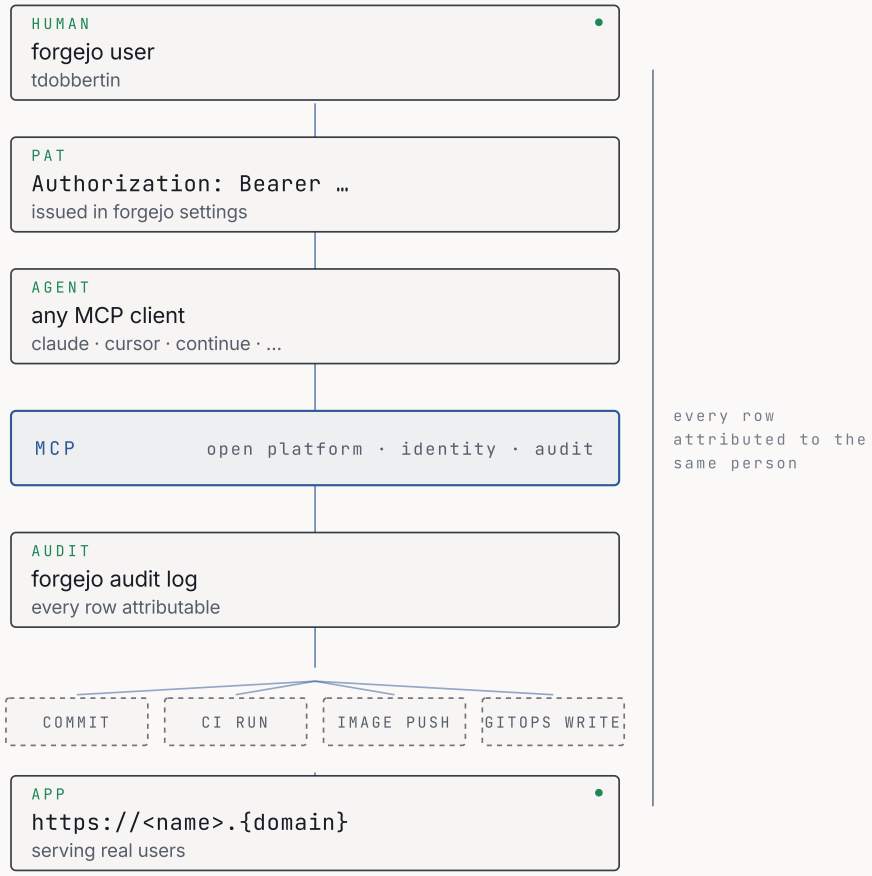

FIGURE 2 — EVERY ARTIFACT TRACES BACK TO A HUMAN. THE AGENT ACTS AS THE HUMAN, NEVER ALONGSIDE THEM; THE AUDIT LOG IS A CHAIN OF ROWS ATTRIBUTED TO THE SAME PERSON.

This is a deliberately small idea with a deliberately large effect. Most platforms eventually grow service accounts, machine users, bot identities — the soft-edged actors that make audit trails ambiguous. ("This commit was made by `ci-bot`. Who is `ci-bot`?") The Open Platform refuses to grow them. The CI runners use ephemeral, attributed credentials. The provisioner Jobs run with namespace-scoped service accounts and write Secrets without an outgoing identity. The deploys are signed not by Flux but by the human whose commit triggered them. Every row in every log is attributable to a person you could send an email to.

The practical consequence is that the question *"who built this app?"* always has a precise answer — at the level of the repository, the commit, the deploy, and the running pod. The same applies to *"who authorized this change?"*, *"who can revoke it?"*, and *"who is responsible if it breaks?"*. Those four questions are the substance of trust in software supply chains; on most platforms they fan out across vendor consoles. Here they collapse to a single Forgejo username.

> *The audit trail is not an additional feature. It is the only mechanism by which the work can happen at all.*

The arrangement also bounds what agents can do. An agent without a tethered human PAT cannot create an app, cannot push code, cannot deploy. An agent acting under a non-admin user's PAT can do everything that user can do, and nothing more. The platform never lets the agent expand its own surface; the surface is whatever the human's identity says it is.

The golden path: from `create_app` to a URL

The platform exposes a small, deliberate set of operations through its MCP server. The most important is `create_app`. Given an organization, a name, and a one-line description, it does the following — atomically from the caller's perspective, asynchronously under the hood:

```
create_app(org="my-org", name="widgets", description="Widgets inventory")
  ⟶ { workflow_handle, app_url }
```

A workflow runs. It generates a new repository in the chosen organization from the platform's app template. It writes a GitOps entry under `gitops-system/apps/`. Flux picks that entry up on its next reconciliation cycle. A provisioner Job creates a Postgres database, a MinIO bucket, and a Forgejo OAuth application, and writes the corresponding `app-db`, `app-s3`, and `app-oauth` Secrets into the new namespace. Forgejo Actions builds the first container image, pushes it to the in-cluster registry, and Flux rolls out a Deployment. Traefik begins serving traffic at `https://widgets.{domain}`. Total time from call to working URL: roughly two minutes.

FIGURE 3 — THE GOLDEN PATH. ONE MCP CALL LEADS TO A REAL REPOSITORY, A REAL DATABASE, A REAL CONTAINER, A REAL URL — WITH EVERY STEP ATTRIBUTABLE IN THE AUDIT LOG.

What ships in that first repo is itself the next interesting thing about the platform. Every app starts from a single template that includes a working *items inventory* example — list page, detail page, CRUD API routes, schema, tests. The example is not the point; it is the proof that the substrate works end-to-end. The first job of any new app is to delete the example cleanly and replace it with whatever the actual app does. The template's own documentation calls this the *pivot*: the moment the app stops being a generic CRUD demo and becomes itself.

Code is not the only artifact the golden path produces. Before code is written, the platform asks for planning via a `plan_work` call that enforces a structured Forgejo issue with four fields:

context, *acceptance criteria*, *files likely touched*, *test plan*. The four fields are non-negotiable and were chosen because they correspond to the things that, in practice, reduce review churn. A PR opened against an issue that names its acceptance criteria can be reviewed in minutes; one opened against "do the thing" cannot. The agents that build apps on this platform are required to plan in this format. The reviewers, human and otherwise, are entitled to refuse work that does not.

Strong foundation, not a constraint

The most common worry about shared templates is that they produce cookie-cutter apps. The platform's evidence to the contrary is its own running apps.

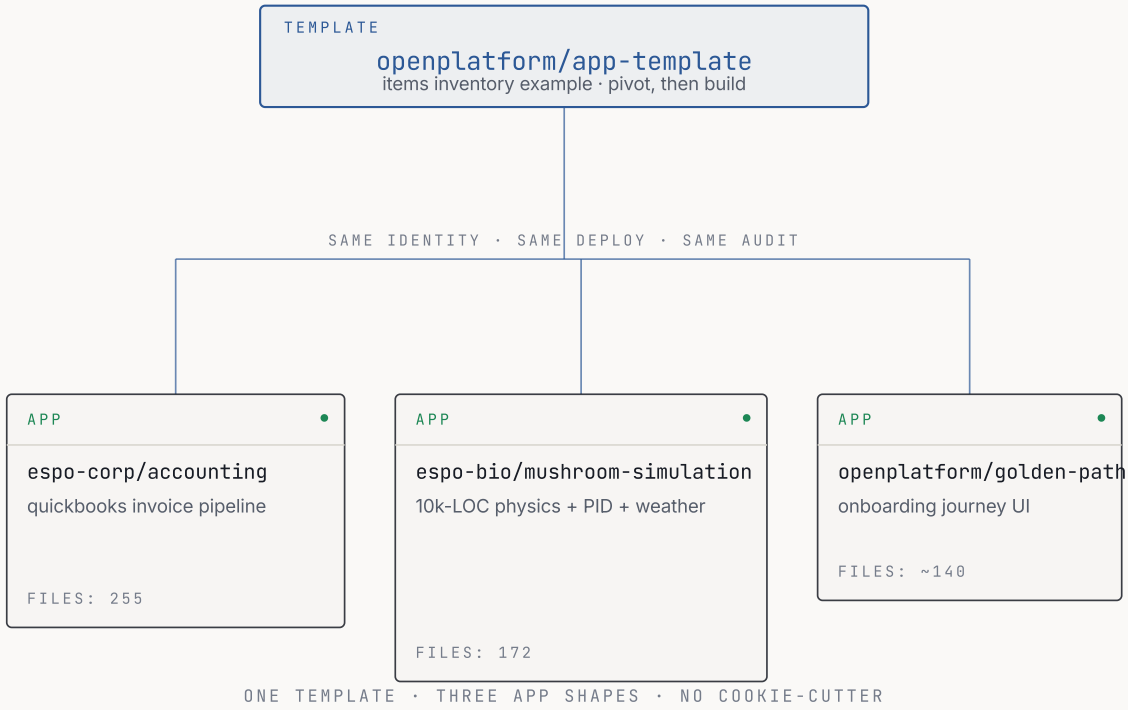

FIGURE 4 — THREE APPS, ONE TEMPLATE. SAME AUTHENTICATION, SAME DEPLOY PIPELINE, SAME AUDIT TRAIL. WILDLY DIFFERENT SHAPES.

Consider three of them, all built from the same template, all running on the same cluster:

`espo-corp/accounting` is an operational tool. It receives pre-computed invoices from upstream engineering systems via `POST /api/v1/invoices`, posts them to QuickBooks Online, archives the rendered PDFs, and surfaces exceptions — re-syncs, voids, catalog reviews, OAuth refreshes — on an admin dashboard. It has more than two hundred source files, twenty-plus test suites, a custom workflow library, a QuickBooks client, and a deep admin component tree. It looks nothing like a CRUD demo.

`espo-bio/mushroom-simulation` is a technical specialist. It runs an interactive cultivation simulation modeling mushroom growth, HVAC equipment state, cumulative energy cost, and carbon footprint. It pulls historical Chicago weather data (ORD TMY3, NREL/TP-581-43156) and

posts it through a ten-thousand-line simulation engine, a PID controller, and a cost model. The user interface is three pages of sliders and live-recomputing charts. It looks nothing like a CRUD demo and nothing like an accounting tool.

`openplatform/golden-path` is a platform-meta app. It is the canonical *zero-to-running* onboarding journey itself — the UI that walks a new user through creating an organization, creating an app, and watching it come up. It is also the reference implementation of the audit story, surfacing the chain of events that the natural-person tether produces.

The three apps share an authentication system. They share a CI pipeline. They share a deployment topology. They share an observability surface. They share the fact that every change to any of them is attributable to a Forgejo user. And they share nothing else — not their data models, not their user interfaces, not their domain language, not their files-per-app, not their test counts. The template is a contract about identity, deployment, and audit. It is not a contract about what the app does.

This matters because the alternative — a platform that constrains app shape — is the kind of platform that gets thrown out the first time a real business problem requires something the platform did not anticipate. The Open Platform does not anticipate. It provides the boring substrate and gets out of the way.

Artifacts, not transcripts

When an agent uses a generic chat interface to "help with code," its natural unit of output is text. The user reads the text, decides which parts to keep, copies them into files, runs them, debugs them. The agent's effort produces a transcript; the human turns the transcript into software. Most of the effort is in the turning.

On the Open Platform, an agent's natural unit of output is not text. It is a Forgejo issue with the four required planning fields. Then it is a branch. Then it is a multi-file atomic commit, written through `batch_update_files` so the index never sits in a half-converted state. Then it is a pull request with `closes_issue` set, with green CI, with a live preview environment at a predictable URL. Then it is a merge, a build, and a deploy. Every one of those is a real artifact: a row in a Git log, a row in an audit log, a row in a CI history.

> *The agent does the typing. The platform does the remembering.*

The shift from transcript-as-output to artifact-as-output is the entire reason the platform tolerates — even encourages — agents to "spam tokens." On a platform whose deliverable is text, every token is overhead. On a platform whose deliverable is artifacts, the token is the medium, but the asset is the deployed app, the auditable diff, the testable preview environment. The agent can be as profligate with prose as it wants. What lands in the repository is what matters; the rest evaporates.

The same shift makes review tractable. Reviewing a transcript means re-running the agent's reasoning in your head. Reviewing an artifact means reading a diff, running a preview, and clicking through a checklist. The latter is what software engineers already know how to do.

Human-in-the-loop and the review floor

Every PR on the platform is required to clear the same review standards before merge. The standards are not policy in the sense of a wiki page that nobody reads; they are mechanical checks, encoded in the platform's docs, enforced by reviewers, and structurally hard to skip.

The non-negotiable items are short and specific:

Parameterized queries. Never string-interpolate values into SQL — not for arguments, not for IN-clauses, not "just this once." Sortable columns flow through an allow-listed `SORTABLE` map; raw search params never become part of a SQL statement.

Validation in the handler. Zod schemas validate every mutating request. The validator must match the database schema, including length and nullability constraints.

Session checks in the handler, not middleware. Mutating routes (`POST` , `PATCH` , `DELETE`) short-circuit with 401 if the session is absent. Middleware-only enforcement is rejected because middleware is removed and added with too little ceremony to be load-bearing for authorization.

Transactional mutations. Any multi-statement state change is wrapped in an explicit `BEGIN` / `COMMIT` . A half-applied change is not allowed to land on the floor.

Live probes for cross-boundary changes. Any handler that calls Forgejo, Kubernetes, or Postgres is verified against the running system *before merge*, not after deploy. The platform docs are explicit that "treating prod as the sandbox once cost a revert plus a platform-stack RBAC PR" — a lesson learned the hard way and codified into the review checklist.

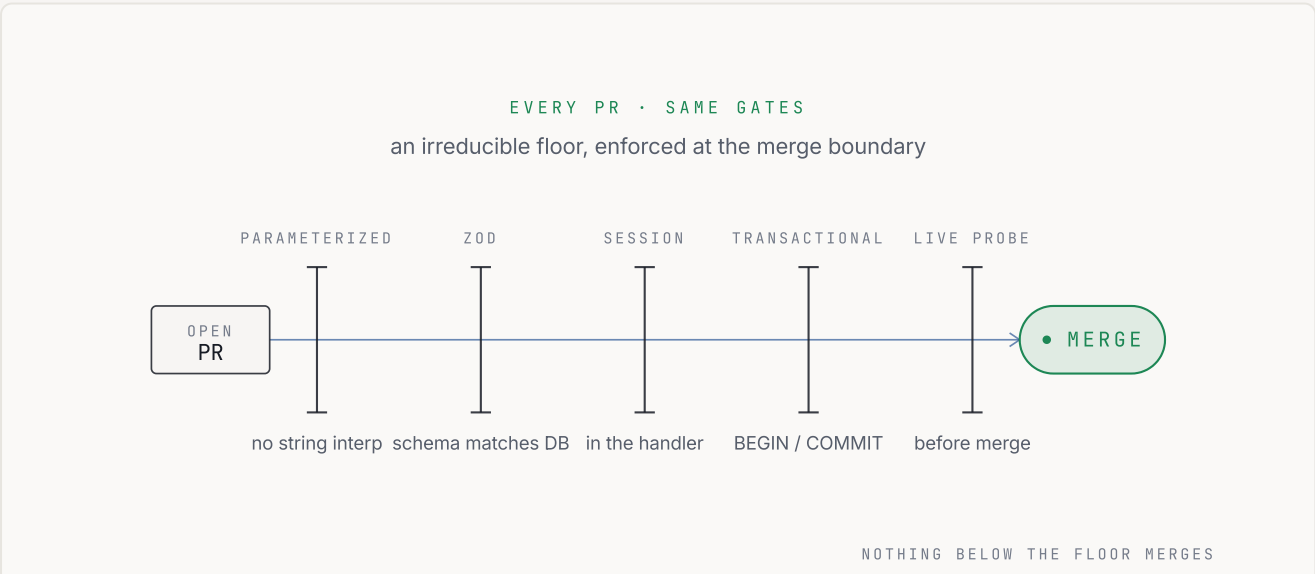

FIGURE 5 — THE REVIEW FLOOR. EVERY PULL REQUEST, WRITTEN BY A HUMAN OR BY AN AGENT, CLEARS THE SAME GATES BEFORE IT CAN MERGE.

The point of the list is not that it is unusually long; it is that it is short. The platform's claim is not "we have comprehensive review standards." It is "we have an irreducible floor, and nothing below the floor merges." Hallucinated SQL doesn't merge. Forgotten session checks don't merge. Half-transactions don't merge. Code that passes typecheck and tests but interacts with Forgejo or Kubernetes in a way that has never been probed live doesn't merge.

The floor lives at the merge boundary, where it can be enforced once and reused for every contributor. Agents working on the platform produce code that meets the floor or it does not land. Humans working on the platform produce code that meets the floor or it does not land. The two cases are indistinguishable.

Empowering non-technical users

The most consequential implication of the model is one the architecture diagrams do not show. A non-technical user — an operator, a domain expert, a founder without a development team — equipped with a Forgejo PAT and an agent client can describe what they need, watch the agent call `create_app`, and end up with a deployed, audited, identity-aware application bound to their name.

The agent does not have to be ours. Claude Code, Cursor, Continue, Windsurf, VS Code with an MCP client — all of them work, because the platform exposes itself over standard MCP and standard HTTP and does not care which client is on the other end. The agent is doing the typing. The platform is doing the substrate. The operator is doing the part nobody else can do, which is knowing what the business needs.

The historical alternative — "tell the developer what you want, watch them translate it through a ticket system, wait, get back something that wasn't quite what you meant" — is not somebody's idea of healthy collaboration. It is a structural artifact of the developer being the only person able to talk to the substrate. Once an agent can talk to the substrate on the operator's behalf, the developer becomes a reviewer rather than a translator. That is not a small change.

Observability today, observability tomorrow

A credible platform document names what is not there.

The observability surface that ships with the Open Platform today is operational, not deep. The platform provides:

- per-app status and health (`get_app_status` , `check_app_health`)
- per-app logs, including previous-crash logs (`get_app_logs`)
- per-workflow CI history (`list_workflow_runs`) and per-run logs (`get_workflow_run_logs`)
- a Kubernetes dashboard at `https://headlamp.{domain}` with OIDC against Forgejo
- a MinIO console for object-storage inspection
- the Forgejo Actions UI for raw CI artifacts
- Flux's own reconciliation state (`flux get kustomizations` , or via Headlamp)

It does not, today, ship Prometheus. It does not ship Grafana. It does not ship distributed tracing. It does not document an autoscaling story for individual apps, a secret-rotation procedure, a backup/restore workflow, or a multi-cluster topology. Those absences are real, and listing them is not a marketing exercise; it is the only way to talk honestly about what the platform is.

What substitutes today is the audit trail itself. Every change to every app is a Forgejo commit attributed to a human. Every deploy is a Flux reconciliation logged against the GitOps repository. Every pod's stdout is reachable through `get_app_logs` . Every CI run is reachable through `get_workflow_run_logs` . The union of those four streams answers, in practice, almost every incident question a small operations team needs answered. It does not answer the SLO question; it does not answer the tail-latency question; it does not answer the distributed-trace question. Those are the questions the next version of the platform is being designed to answer.

The choice to ship operational observability before infrastructural observability was deliberate. Logs and status reach an operator first. Metrics, traces, dashboards reach an operator second. The first hundred apps benefited more from the first set than they would have from the second.

Sovereign by default

Every component named in this paper runs on the same Kubernetes cluster.

That includes Forgejo, Flux, Traefik, Postgres (via the CloudNativePG operator), MinIO, the in-cluster container registry, the platform's MCP server, the Headlamp dashboard, and every workload app. There is no SaaS dependency for identity, no SaaS dependency for source control, no SaaS dependency for CI, no SaaS dependency for storage, no SaaS dependency for routing, no SaaS dependency for the audit trail.

This is not an aesthetic choice. It is the reason the audit trail can be trusted at all. Each external boundary an audit chain crosses is a place where the chain could be broken or selectively logged. A platform whose identity provider lives in a vendor account inherits that vendor's notion of who acted; a platform whose CI runs in a vendor's runners inherits that vendor's view of what ran. The Open Platform does not inherit those views. The view is the platform's own, end to end, and it is exactly as trustworthy as the cluster the user controls.

The same property is what makes "bring your favorite agent" coherent. The agent is on the *outside* of the platform; the platform never depends on the agent vendor's continued goodwill, pricing, or availability. An organization that adopts the Open Platform and runs it for five years can swap its preferred agent — or remove agents entirely — without the platform noticing.

Scale: what we've learned from 100+ apps

Numbers are easier to trust than narratives, so:

At the time this paper was written, the platform was hosting more than a hundred deployed applications across fifteen organizations. Roughly ninety-five percent of those applications were running; the small remainder were intentionally stopped — paused for cost or pending migration. Every one of those apps was reachable at a deterministic URL derived from its organization and name. Every one was the result of a `create_app` call, in some cases by a human typing into an MCP client, in some cases by an agent acting on a human's behalf.

Three observations from running that population:

The namespace convention scales mechanically. Every operational tool that addresses an app — for logs, for status, for health — does so by deriving the namespace from the `(org, repo, pr?)` triple. We have added apps, removed apps, renamed orgs, opened previews, and never had to update the tools. The same pattern that holds for a corporate accounting app holds for a fifteen-thousand-line physics simulation in the same cluster.

Preview environments are not a special case. Every PR on every app repo gets a preview environment automatically: a fresh namespace, a fresh database, a fresh bucket, a fresh OAuth application, a fresh URL. The preview-vs-production split is the only thing the platform models specially, and it is modelled by appending `-pr-<N>` to the namespace and rerunning the provisioner Job. Nothing in the application code changes to support previews; the platform does the work.

Diversity is the point. The hundred-plus apps include accounting tools, CAD ingestion pipelines, electrical schematic editors, mushroom-cultivation simulators, recruiting funnels, applicant tracking systems, multiplayer arena games, internal dashboards, marketing sites, and a half-dozen apps whose purpose is to test or monitor the platform itself. They share a deployment topology and no domain. That is the right ratio.

Vl, V2, and what comes next

The platform described in this paper is the first implementation. It is real — running real apps, for real users, on real infrastructure — and it is also not the final shape. A second implementation is in design, informed by exactly the gaps named in the observability section: a richer metrics and tracing surface, a documented autoscaling story, a documented secret-rotation procedure, a documented disaster-recovery workflow.

V1 will not be retired. The lessons that produced V2 came from operating V1 at scale; the architectural commitments of V1 — the four "ones", the natural-person tether, the artifact-not-transcript principle, the review floor — are the things V2 is meant to keep. What V2 adds is not a replacement of the foundation but a thicker layer of operational tooling on top of it.

The honest framing for any reader is that V1 is what you can adopt today, and V2 is what is being designed for the readers who, after using V1 long enough to feel its edges, would like the next version of the operational surface. The pattern is the same one V1 itself followed: ship something real, learn from its operation, design the next version against the gaps.

Closing: what to do next

The most useful thing a technical reader can do, when the platform's Forgejo opens to the public, is read three repositories in order:

- `openplatform/golden-path` — the onboarding journey itself; the canonical *zero-to-running* narrative implemented as a real app.
- `espo-corp/accounting` — an operational, business-critical tool with deep customization beyond the template.
- `espo-bio/mushroom-simulation` — a technical specialist app whose codebase looks nothing like a CRUD demo.

Three repositories. One template. Three completely different shapes. About thirty minutes of careful reading, which compresses the argument of this paper into its empirical form.

A secondary, lower-effort follow is the LinkedIn account for V2 progress and platform updates. It is the right place to follow if the paper is interesting but the time-to-adoption is longer.

The platform is, intentionally, boring. The argument of this paper is that the boredom is the feature. The interesting work is now happening on top of platforms like this one; the platforms themselves can — and should — get out of the way.

Appendix A — References & artifacts

Platform documentation slugs (read via the platform's MCP `get_doc` tool):

- `02-mental-model` — the four "ones" verbatim.
- `06-first-app` — the canonical golden-path sequence.
- `07-previews` — preview-environment semantics.
- `08-operating` — operational observability surface.
- `12-golden-path-planning` — the four-field issue planning contract.
- `13-review-standards` — the review floor.
- `14-ai-services` — Ollama + ChromaDB integration for AI-native apps.

The four-field issue template (required by `plan_work`):

- *Context* — what is the situation that makes this work necessary?
- *Acceptance criteria* — testable bullets written as present-tense assertions.
- *Files likely touched* — scope sketch, not a contract.
- *Test plan* — concrete steps to verify acceptance, before review.

Tooling integrations. The platform's MCP server has been verified against Claude Code, Cursor, Continue, Windsurf, and VS Code with an MCP client. Adding a new client requires no platform-side change.

Authors. This paper was assembled by the ESPO Open Platform team in May 2026. Corrections and questions: open an issue against `espodev/www` once the Forgejo instance is public, or reach the team via the portal at `www.espodev.com/portal`.