TECHNICAL CASE STUDY · CODEFRONT LABS

ForgeLoop

An engineering control plane that makes coding agents accountable for verified work.

How ForgeLoop turns a labelled GitHub issue into a validated task graph, runs bounded agents on infrastructure the user controls, and only delivers a pull request when configured checks and every acceptance criterion have recorded, checksummed evidence for the exact commit under review. This document covers the architecture, the planning and scheduling invariants, the guarded agent loop, verification and evidence integrity, delivery policy, the October security audit, a recorded unattended issue-to-merge run, and the launch gates that remain open.



Runner 1.0.14 · control plane at commit 4bd9b2f

Prepared 14 November 2026

Contents

01 Executive summary

02 The problem

03 Design principles and constraints

04 System architecture

05 The delivery lifecycle

06 Planning, scheduling and repair

07 The runner and the guarded agent loop

08 Verification and evidence integrity

09 Delivery policy and auto-merge

10 Security hardening

11 Proof: an unattended issue-to-merge run

12 Operations and reliability

13 By the numbers

14 Limitations and what comes next

15 Lessons learned

Sources: the ForgeLoop repository (`JRH89/ForgeLoop` , 472 commits on `master` between 18 September and 28 October 2026), its design and operations documents under `docs/` , the October 9 validation record and October 26 security audit, the project notes in the Codefront Labs vault (including the proposed "Prompt to Project" plan), and the public site at forgeloop.codefrontlabs.com. Code metrics were measured from the working tree on the preparation date. Items that are planned or unverified are labelled as such; no customer, adoption or revenue figures are claimed.

1. Executive summary

ForgeLoop is a hosted software-delivery product built around a single rule: **an agent saying "done" is not evidence.** A developer labels a GitHub issue. ForgeLoop plans it into bounded tasks, has a coding agent implement them on a runner the developer controls, runs the repository's real checks in pinned containers, reviews the diff independently, and opens a pull request — or merges it, if the organization has explicitly allowed that.

~43k
lines of Java and TypeScript across five modules

900+
automated unit and integration tests, plus Playwright E2E

$0.22
recorded model cost for an unattended issue-to-merged-PR run

12
runner packages per release across Windows, macOS and Linux

The hosted control plane (Java 21, Spring Boot, GraphQL, PostgreSQL) coordinates runs, policy, leases, membership and the audit trail. It never checks out the customer's repository and never holds their model keys. The runner — a Java desktop and CLI process — claims tasks through expiring leases, works in isolated Git worktrees, calls the model provider with a local credential and uploads checksummed evidence through its active lease.

> **Outcome.** In about six weeks of solo development, ForgeLoop went from an empty repository to a live hosted product with signed-in self-service onboarding, published cross-platform runner previews, a public replay of a recorded run, and a validated unattended delivery: issue #10 on a public repository became merged PR #11 with no human approval recorded.

2. The problem

By late 2026, coding agents could reliably write a plausible patch. Writing the patch, though, is one stage of software delivery. Everything around it was usually left to the person running the agent:

- **Scope.** Which issue is eligible, which files the agent may touch, and how much it may spend.
- **Coordination.** When a change spans backend and frontend, two agents (or two machines) need ownership boundaries and an integration step, not a race.
- **Proof.** A completion summary is the model's own claim. Reviewers need test output tied to the exact commit, and coverage of every acceptance criterion.
- **Recovery.** Workers crash, leases expire, checks fail. Retrying without limits burns money; giving up silently loses work.
- **Trust boundary.** Source code and provider keys are sensitive. Many teams will not hand both to a hosted service.

> **Goal.** Make every one of those concerns an explicit, persisted, testable part of a working product, so the path from issue to merged pull request is auditable end to end and fails closed whenever evidence is missing.

ForgeLoop began as a ticketing-SaaS scaffold. The first roadmap phase ("reset the product boundary") moved that application into a separate repository, Ticketly, which then became just one external repository ForgeLoop delivers into — with no special runner logic. That decision kept the scheduler honest: nothing in the control plane knows what kind of application it is changing.

3. Design principles and constraints

Separate coordination from execution

The control plane owns delivery state; the runner owns repository commands, model calls and secrets. Neither can do the other's job.

Verify a specific change

Gate output, criterion coverage and delivery checks are tied to one commit SHA. A changed branch never inherits earlier evidence.

Bound everything

Attempts, tokens, wall time, money, tool calls, output size and changed paths all have fixed limits recorded in a policy snapshot.

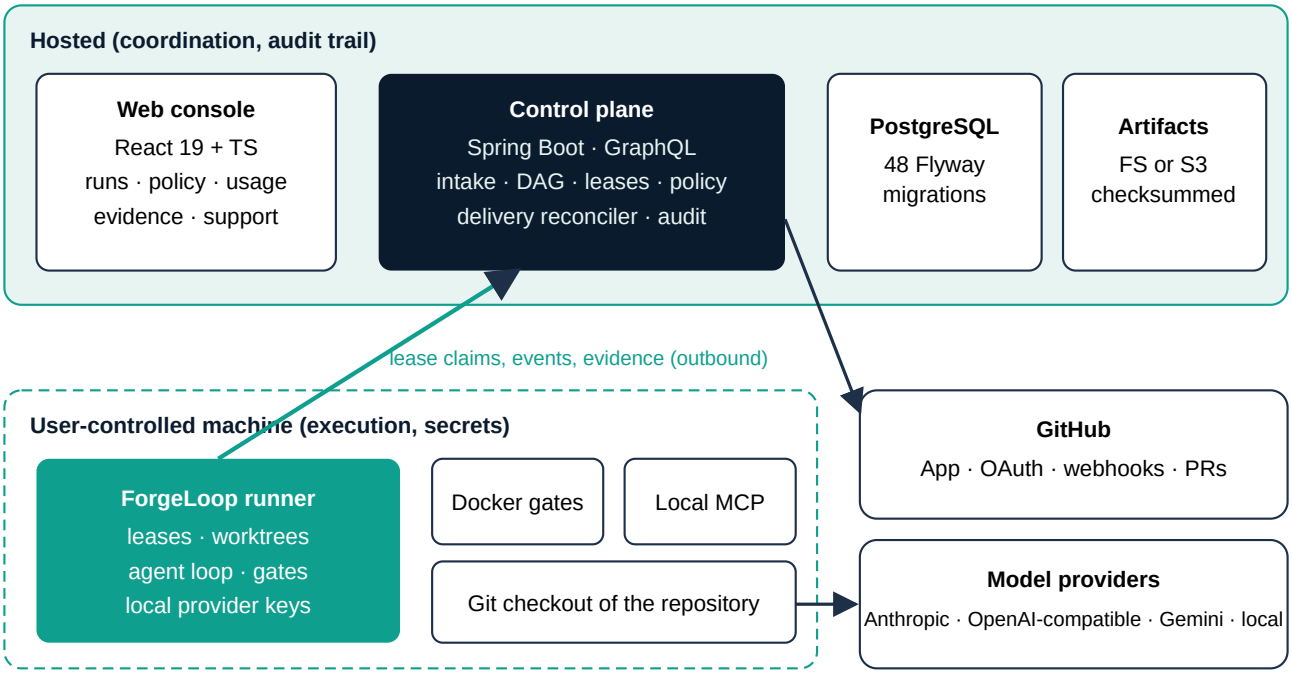
Fail closed

Malformed plans, unknown policy, exhausted repair budgets and legacy runs without a valid snapshot all stop, block or require a human — never succeed.

Hard constraints

Constraint	Consequence in the design
Repository access requires explicit authorization	GitHub App installation, organization membership and roles; intake labels, optional assignee gates and per-repository bypass are opt-in.
Credentials stay on enrolled infrastructure	Provider keys live in runner-local secret storage; the control plane sees only capability metadata and usage totals.
Verification commands cannot come from a task or an operator at run time	Administrators configure a versioned `RepositoryPolicy`; each run snapshots it and records the revision.
Unknown cost is not zero cost	Prices that are not configured remain explicitly unknown in budgets and usage views; nothing is guessed.
Worktrees are not a sandbox	Documented as checkout isolation only; real containment comes from pinned containers, path guards and hardened Git.

4. System architecture

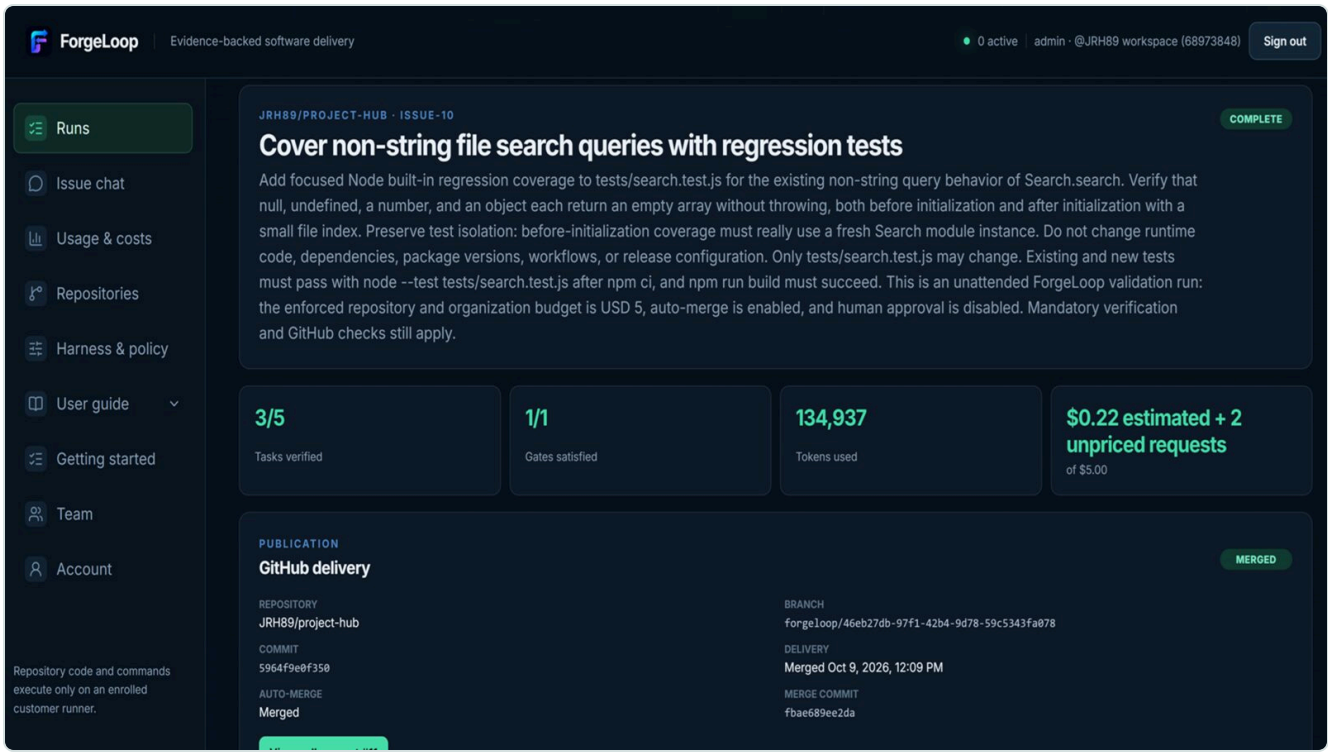


Module	Responsibility
`control-plane`	Spring Boot GraphQL API: organization-scoped state, intake, task graph persistence, scheduling, runner leases, policy, budgets, audit, GitHub coordination and the verified-delivery reconciler.
`runner`	Separately versioned desktop/CLI client: enrollment, heartbeats, lease-bound checkout, worktrees, provider adapters, the agent loop, Docker gates and evidence upload.
`harness`	State machine, evidence writer, model-selection records and worker contracts shared by both sides.
`frontend`	Prerendered public site and the authenticated console. It never makes authorization decisions.
`mcp-server`	Stdio MCP gateway exposing least-privilege run, policy, task, evidence and cancellation operations. It cannot execute repository commands.

The runner only makes outbound connections, so it works behind NAT and firewalls without inbound ports. The hosted stack runs under Docker Compose on a Linux server behind Cloudflare Tunnel, with every host port bound to loopback.

5. The delivery lifecycle

1. **Intake.** A signed GitHub webhook (HMAC-SHA256, constant-time compare) arrives. Delivery IDs are claimed as idempotency locks and released on failure, so a redelivery is processed exactly once. Repository rules — intake labels, optional assignee gate — decide eligibility. Issues can also start in **Issue Chat**, where an AI helps refine an editable specification before it is published to GitHub.
2. **Planning.** A non-writing planner returns acceptance criteria and a task graph (section 6).
3. **Execution.** Runners claim compatible tasks and implement them in isolated worktrees (section 7).
4. **Integration.** Declared dependency commits are cherry-picked in graph order onto a detached worktree without a shell.
5. **Verification.** Required policy gates run in pinned containers and must cover every criterion (section 8).
6. **Review.** An independent review stage examines the integrated diff.
7. **Delivery.** The snapshotted approval policy decides between draft PR, ready PR, or guarded auto-merge (section 9).



Run detail: verification counters, recorded usage and GitHub delivery status for one captured run. Figures belong to that run only.

6. Planning, scheduling and repair

The planner contract

The planner must return one JSON object with exactly `acceptanceCriteria` and `tasks`. Each task declares a stable key, a role, a required runner capability, dependency keys, repository-relative owned path prefixes, an attempt budget of one to five and a budget in micro-dollars. The runner validates shape, path safety, keys, roles, dependencies, cycles and total budget, and the control plane *independently repeats* those checks inside one transaction. A malformed, cyclic, over-budget, traversal-bearing or partially valid plan is rejected without any task becoming visible to dispatch.

Scheduling invariants

A runner receives a task only when its capability matches exactly, every dependency is in an allowed completed state, neither task nor run has exhausted its known budget, and none of its owned paths overlaps an active task in the same run.

> **Race-free claims.** A claim locks every task row of the run in a stable order before re-checking eligibility. That serializes competing runners and prevents two of them from entering overlapping paths, while disjoint backend and frontend tasks stay independently claimable and run in parallel.

Bounded repair

A failed or expired lease increments only its owning task's attempt counter. With attempts left, the task becomes `REPAIR_QUEUED` and is re-dispatched to a `REPAIR` role with a deliberately small package: failure category, change SHA, evidence digest, owned paths and criteria. Raw prompts, source files and unrestricted logs are not persisted in it. When the next failure would exceed the budget, the task becomes `FAILED` and the run `BLOCKED` — never successful, and no other task is silently reset.

7. The runner and the guarded agent loop

The runner enrolls with a one-time token; credentials are `SecureRandom` values stored server-side only as SHA-256 hashes. Browser pairing lets an administrator approve a desktop runner from the console. Heartbeats, capability matching and pause/recovery let the scheduler route work to the right machine.

A tool surface with no shell

The multi-turn writing worker (behind a default-off runner switch plus a repository policy opt-in) talks to providers through provider-neutral, append-only conversations and exposes exactly seven tools:

Tool	Bound
`list_files` / `read_file` / `search_files`	500 entries; 1 MiB UTF-8 in 1–400 line pages; safe-bounded regex over 20,000 files, 100 matches
`write_file` / `edit_file`	Atomic writes up to 500,000 chars, owned paths only; ambiguous edits require explicit `replaceAll`
`run_gate`	Advisory run of a declared gate in the restricted Docker executor; cannot submit official evidence
`finish`	Rejects empty changes, >20 paths, deletes/renames, or any path outside owned prefixes; then commits

There is no shell, delete, rename, install or process tool. Every call passes through a `ToolGateway` that checks the role grant, validates arguments, runs an enforcement chain, retries one transient failure and writes a write-ahead JSONL journal. Tool results entering the conversation are capped at 16 KiB; repository text is treated as untrusted data.

Budgets and stop conditions

Each attempt is bounded by tool calls (1–1,000), tokens (10k–100M), wall time (60 s–4 h) and conversation size. The lease is renewed every 30 seconds; a renewal failure halts further tools and turns, and a lost lease can never publish a commit. Refusal, two consecutive no-tool replies, output truncation, budget, provider and harness failures each have distinct, categorized outcomes.

Runner-local MCP

The control plane may *name* MCP context tools, but the runner applies its own exact-command allowlist, runs the process in the worktree, accepts only text within 15 seconds and 32 KiB, then kills it. Any protocol error fails the lease rather than continuing with incomplete context.

8. Verification and evidence integrity

ForgeLoop never accepts an arbitrary verification command. Each repository policy gate declares a kind (`CONTAINER`, `BROWSER`, `SECURITY`, `CONTRACT`, `COMPOSE`), an immutable image digest, an argv array (no shell expansion), an explicit network decision, a timeout and a coverage rule. The current rule is strict: `ALL` means every acceptance criterion needs passing evidence from that gate.

```
# one required gate in a repository policy (abridged)
{ name: "unit", kind: "CONTAINER",
  imageDigest: "node@sha256:ebfe2f90…",
  command: ["sh","-c","npx jest --reporters=default --reporters=jest-junit"],
  networkPolicy: "EGRESS", timeoutSeconds: 900,
  required: true, criterionCoverage: "ALL", testReport: "JUNIT_XML" }
```

The evidence chain

1. The runner copies the read-only worktree into an ephemeral container filesystem and executes the policy argv with time and output limits.
2. Output is redacted for common credential forms and written as an atomic JSON artifact plus a SHA-256 manifest.
3. The runner verifies the local bundle and uploads the exact bytes through its active, authenticated lease.
4. The control plane recomputes both checksums, performs storage read-after-write verification, rejects metadata that differs from the run's policy snapshot and rejects output that still resembles a raw credential — only then issuing an `artifact://` reference.

> **Only PASSED counts.** `FAILED`, `TIMED_OUT`, `SKIPPED_BY_POLICY` and `MANUAL_OVERRIDE` never make a run `READY_FOR_REVIEW`, so GitHub delivery cannot create a branch or PR for them. Test-first mode goes further: JUnit XML is mounted outside the writable workspace, so a committed `test-results/` directory cannot shadow or forge the report.

9. Delivery policy and auto-merge

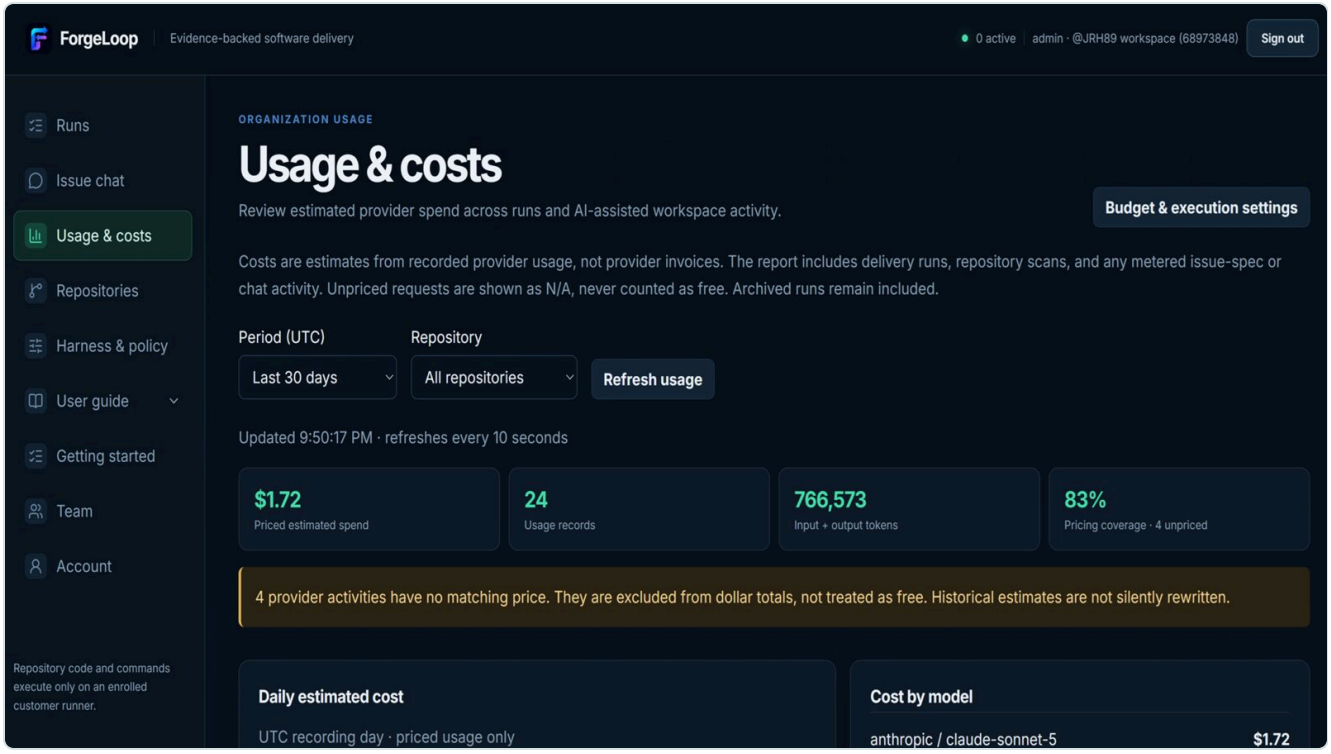
Each run captures a checksummed snapshot of organization policy, so later edits never retroactively change its approval or merge requirements.

Human approval	Auto-merge	After verification passes
On	On	Wait for approval, publish a ready PR, merge after GitHub checks
On	Off	Wait for approval, publish a draft PR for manual delivery
Off	On	Publish automatically, merge after GitHub checks
Off	Off	Publish a draft PR automatically; merging stays manual

Auto-merge is off by default. When enabled it still requires the exact reviewed head SHA persisted at PR creation, success of every GitHub check and commit status, and a merge request that carries that expected SHA — so a push between review and merge aborts the merge. A reconciler re-checks pending branches every 30 seconds; failures stay pending and no approval event is ever fabricated. Legacy runs without a valid snapshot fail closed to human approval.

Cost and budget controls

Organization policy caps run budget, parallelism and permitted providers; repository budgets narrow it further, and submissions above the organization limit are rejected before any work is created. The usage dashboard separates priced estimates from unpriced provider activity rather than presenting a misleadingly complete total.



Usage and costs, with pricing coverage shown explicitly. A captured snapshot, not guaranteed costs.

10. Security hardening

A full audit on 26 October 2026 reviewed authentication, tenant authorization, the GitHub integration, uploads and downloads, rate limiting, the runner sandbox, the web gateway, deployment configuration and all three dependency trees. No committed secrets were found in the tree or history. Selected findings, all fixed:

Severity	Finding	Fix
High	The GitHub App callback trusted the `installation_id` query parameter, so a member of one tenant could bind another customer's unclaimed installation.	Admin-only installs; state binds tenant and operator; ownership proven via `GET /user/installations` with the operator's own token.
High	On Windows, the path guard missed spellings that resolve to `.git` (`.git.` , `GIT~1` , NTFS streams), allowing an agent to redirect Git config.	`WorktreePathGuard` canonicalizes segments the way Git's `core.protectNTFS` does, on every platform.
Medium	Runner Git honoured repository hooks and `core.fsmonitor` .	One hardened prefix for every Git call: hooks to `/dev/null` , fsmonitor off.
Medium	Rate limiting keyed on an unauthenticated header; rotating it bypassed limits and could exhaust the window table.	Charge per peer address first, with a separate window per well-formed runner ID.
Medium	Unbounded GraphQL document cost; public introspection.	Depth 10 / complexity 300, set after measuring all 74 first-party operations (max depth 4, 79 fields); introspection off.
Medium	A read-only viewer could start runs that spend the budget.	Operator role required; tenant check moved before the duplicate check to avoid a cross-tenant oracle.

Lower-severity fixes included POST-only sign-out, security headers on static routes, sandboxed CSP on artifact downloads, webhook claim-and-release, and correct session cookie flags. CI adds CodeQL, a supply-chain workflow and immutable action pins.

11. Proof: an unattended issue-to-merge run

The October 9 validation on the public `project-hub` repository is the clearest evidence that the full loop works without a human in it.

Step	Recorded result
Input	Issue #10: add regression coverage for non-string search queries. USD 5 ceiling.
Approval	Human approval disabled by policy; `approved_at` and `approved_by` both NULL.
Verification	Required `unit` gate PASSED, including Node tests and the production build.
Delivery	PR #11 merged automatically at 19:09:22 UTC, commit `fbae689`; issue #10 closed; reconciliation recorded the merge.
Cost	USD 0.218838 known provider cost. Some telemetry was unknown, so this is not claimed as a complete total.

An earlier run on the same repository (issue #6 → PR #7) merged after manual approval. The public site now includes a **/demo** page that replays the issue #10 run step by step from its recorded evidence rather than from a scripted animation.

> **What this proves, and what it does not.** It demonstrates the end-to-end path on a second repository with a real provider. It does not close the operational launch gates in section 14, and a single run is not a quality or success-rate measurement.

12. Operations and reliability

- **Durable state.** Every workflow transition is persisted; runners publish heartbeats; expired leases are recovered; a restart resumes from the database, not from memory.
- **Backups.** A Compose service writes a custom-format PostgreSQL dump every six hours plus a checksummed archive of evidence objects and a `.complete` marker; health fails if no complete backup exists within seven hours.
- **Restore drill.** A script restores a dump into a network-isolated temporary container and reports record counts; CI runs the same drill against its disposable database. A 26 September recovery confirmed 10 runs and two repository connections survived a rebuild.
- **CI.** Five workflows cover the Java modules, frontend, MCP gateway, Windows setup, a disposable Compose end-to-end environment (database restore, runner pairing, package enrollment, browser tests), CodeQL, supply-chain checks and desktop releases.
- **Releases.** Runner 1.0.14 ships Windows and macOS (x64/ARM64) and Linux DEB, RPM, Arch and portable packages, each with a published SHA-256 digest and download discovery from GitHub Releases.
- **Deployment.** Pushes to a self-hosted Gitea remote trigger a webhook deploy; the stack runs behind Cloudflare Tunnel with a 404 catch-all.

13. By the numbers

472
commits, 18 Sep – 28 Oct 2026

24
distinct days with commits

~37.6k
lines of Java across 609 files

780
JUnit `@Test` methods

126
Vitest, MCP and Playwright cases

48
Flyway migrations

74
measured first-party GraphQL operations

50
design, operations and audit documents

Layer	Technology
Backend	Java 21, Spring Boot, Spring Security, GraphQL, Spring Data JPA, Flyway, PostgreSQL
Web	React 19, TypeScript, Vite, Vitest, Playwright; prerendered public pages with SEO and structured data
Runner	Java 21, Git worktrees, Docker, MCP, provider adapters, native desktop packaging
Tool gateway	TypeScript, Model Context Protocol SDK, Zod
Delivery	GitHub Apps and OAuth, GitHub Actions, Docker Compose, Cloudflare Tunnel, FS/S3 artifact storage

14. Limitations and what comes next

Open launch gates

- Physical server reboot validation, including tunnel recovery and runner heartbeat.
- Encrypted off-host backup retention and a restore drill on a different host, including evidence objects.
- Staging burn-in, external alert delivery and an explicit production go/no-go review.
- Signed Windows/macOS installers and macOS notarization — current packages are clearly labelled unsigned previews.
- The multi-turn agent loop is off by default; offline tests do not establish live provider or deployed Docker behaviour at scale.

Proposed: Prompt to Project

The next product direction, recorded as a proposal on 10 November 2026, extends the same engine from one issue to a whole project. A user describes software in plain language; ForgeLoop drafts a brief, assumptions, milestones, architecture, a cost estimate and a dependency-ordered backlog, then drives each issue through the existing pipeline rather than bypassing its checks. Autonomy, budget, merge and optional deployment (through a provider-neutral adapter) remain explicit per-project policy, and completion reports must list unmet requirements alongside evidence. The recommended approach is orchestration of the existing pipeline over one-shot generation, because it keeps isolation, recovery and traceability. This is planning only; no part of it is implemented.

15. Lessons learned

1. **Evidence is a data model, not a log line.** Binding criteria, gate output and checksums to a commit SHA made "is this reviewable?" a query instead of a judgement call, and made forged or stale evidence detectable.
2. **Validate twice, on both sides of the trust boundary.** The runner and the control plane each re-check plans and evidence. The duplication is cheap; a single trusted validator would be a single point of compromise.
3. **Give agents fewer, sharper tools.** Removing the shell and bounding every write turned whole classes of risk into business-rule errors the model can recover from.
4. **Say "unknown" out loud.** Unpriced models, partial telemetry and unverified launch gates are shown as such. That honesty is what makes the verified parts credible.
5. **Audit adversarially, then measure before limiting.** The highest-impact bugs were in trust assumptions (a query parameter, a Windows path spelling), and GraphQL limits were set from the real operation set rather than guessed.
6. **Keep the product boundary clean.** Moving Ticketly out on day one meant the scheduler never learned anything about the code it changes — which is exactly why it works on a second repository.

ForgeLoop is built and operated by Codefront Labs. Product: forgeloop.codefrontlabs.com · Demo: forgeloop.codefrontlabs.com/demo · Source: github.com/JRH89/ForgeLoop