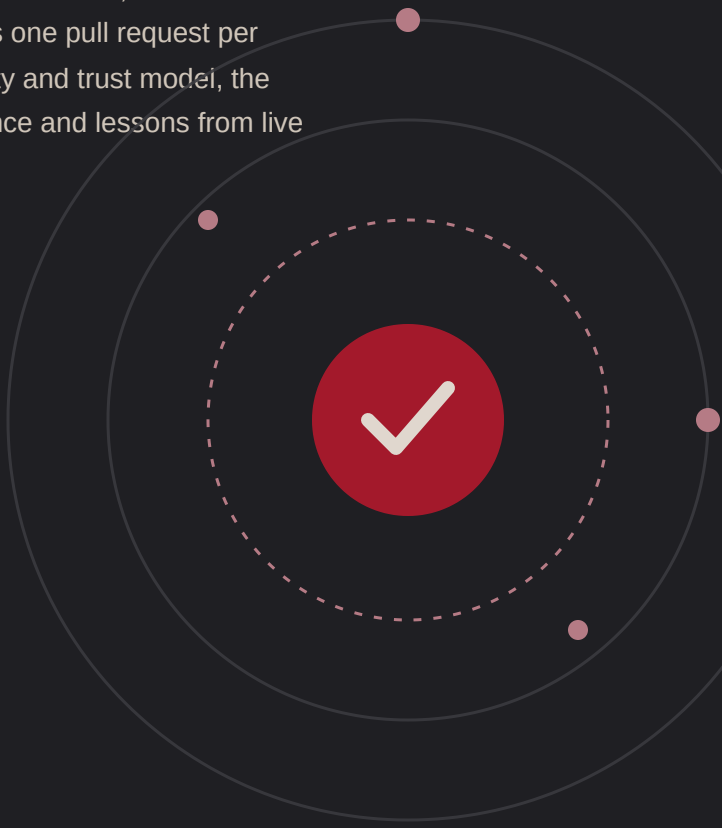
TECHNICAL CASE STUDY · CODEFRONT LABS

Mallace

From an AI red-team prototype to a paid, test-proven security autofix product in six days.

How Mallace finds security bugs in GitHub repositories, proves each one with a regression test that fails on the current code in a no-network sandbox, writes the smallest fix, has an independent model review it, and opens one pull request per verified fix. This document covers the architecture, the safety and trust model, the commercial design, the release engineering, and the evidence and lessons from live acceptance runs.



Release covered: 0.1.0a31 (Python package `mallace`)

Prepared 14 November 2026

Contents

01 Executive summary

02 The problem and the product

03 Origins and timeline

04 System architecture

05 The find → prove → fix → review → deliver pipeline

06 Detection: taint-tracked rules across four languages

07 Safety and trust model

08 Commercial architecture: licensing, metering and payments

09 The read-only website check

10 Release engineering and evidence discipline

11 Live acceptance results

12 Defects found by running it for real

13 The honest miss: the first budgeted AI campaign

14 By the numbers

15 Limitations and what comes next

16 Lessons learned

Sources: the Mallace source repository (155 commits on `main` between 5 and 13 November 2026), its planning and evidence documents under `docs/`, and the project's product status, design notes and implementation journal. Statements about live behaviour are taken from recorded acceptance receipts and status notes; capabilities described only in plans are labelled as such.

1. Executive summary

Mallace is a security tool for GitHub repositories. Most static analysers stop at a list of suspected bugs and leave the developer to sort true from false positives. Mallace goes further. It treats each suspected bug as a claim to be proved. An AI model writes a regression test. The test must *fail* on the current code inside a sandbox with no network, or Mallace leaves the code alone. Only then does a model write the smallest fix. The same test must pass and the code must compile. A second, independent model reviews the change, and a rejected fix goes back to the writer with the reviewer's reasons. Each fix that survives arrives as its own pull request, with the proving test included. Nothing merges automatically.

6 days
first commit to paid product (5–10 Nov 2026)

31
alpha releases (0.1.0a1 → 0.1.0a31)

4
languages with prove-and-fix: Python, JS/TS, PHP, Ruby

~1,250
automated tests across engine, integration, license server and site

The product ships three ways that share one license key: a Python CLI on PyPI, a composite GitHub Action (`JRH89/mallace-action@v0`) and a hosted web app at `mallace.codefrontlabs.com/app` that drives the Action for users who would rather click than type. Scanning is free and unlimited. Fix generation is metered through a license service, with a Free plan ($1 of AI a month) and a Pro plan ($29/month), billed through live Stripe. A separate `mallace site` command runs a read-only, ownership-gated security check of a live website.

What makes it technically interesting

- **Proof before repair.** An exit-code contract (`77` = assertion failed, so the weakness is shown; `0` = passes; `2` = test unusable) is enforced by trusted harnesses in four languages. Model output never decides whether a bug is real.
- **Fail-closed everywhere.** Skipped, failed or unknown checks never count as passing. Budgets default to zero. Grants expire. Ambiguous external writes are reconciled, never blindly retried.
- **Enforcement by architecture, not by obfuscation.** The prompts and model calls live only on the server. The published client cannot write fixes without a valid key, even though its source is readable.
- **Evidence discipline.** Every milestone claim is backed by a recorded receipt (CI run IDs, wheel hashes, PR numbers, spend). Failures are recorded as failures, including a real evaluator miss.

> **Status.** Mallace is a *paid alpha*. Stripe is live and the find-and-fix path has passed live acceptance on owned fixture repositories, but the product is not production-certified, has narrow bug-class coverage, and runs its backend on a single server.

2. The problem and the product

The problem

Static application security testing (SAST) tools are good at producing findings and bad at producing confidence. Teams triage long lists, many of which are constant-only queries, unreachable sinks or idioms the tool doesn't understand. Recent AI autofix products shorten the path from finding to patch, but a generated patch for a false positive is worse than no patch. It adds review load and erodes trust. The gap Mallace targets is the step in between: **demonstrate that the finding is real, on this code, before spending anything on a fix.**

The product, as shipped in 0.1.0a31

Capability	Behaviour	Mode
Prove and fix	SQL injection, command injection, code injection (`eval` / `exec` / `instance_eval`), unsafe deserialization (YAML, pickle, PHP `unserialize` , Ruby `Marshal`). 17 rules across Python, JavaScript/TypeScript, PHP and Ruby, including Django, Flask, FastAPI, Express, Next.js, Laravel, WordPress and Rails idioms.	Metered (license)
Leaked secrets	Nine high-confidence key patterns over the scanned revision. The value is never printed.	Free, report only
Vulnerable dependencies	Exact pins in `requirements*.txt` , `package-lock.json` , `composer.lock` , `Gemfile.lock` checked against OSV. Only names and versions leave the machine.	Free, report only
Website check	Read-only HTTPS, header, cookie, CORS, exposed-file and version checks after DNS/file/meta ownership proof.	Free
Delivery	One issue and one PR per verified fix, plain-English explanation, the regression test, and Action annotations on affected lines. Findings that already have an open Mallace PR are skipped.	Never auto-merged

Three surfaces, one engine

CLI

`pipx install --pre mallace` , then `mallace doctor` , `mallace fix --scan-only` and `mallace fix --budget 5` . It runs inside any GitHub checkout and needs no files written. The repository comes from `origin` , and the revision is the pushed base branch, so unpushed work is never scanned or patched.

GitHub Action

A composite action. It installs from PyPI, unshallows and fetches the base branch, runs fix or scan, and writes annotations, outputs (`findings` , `fixed` , `pull-requests`) and a job summary. No local Docker is needed.

Hosted app (/app)

Next.js on Vercel. Users sign in through the *Mallace Security* GitHub App, and the user token is kept only in an AES-GCM-sealed HttpOnly cookie for 8 hours. The app opens a setup PR that adds the workflow, walks the user through the license secret and Actions permissions, and reads runs, annotated findings and fix PRs back from GitHub. The site has no database of its own.

Who it serves

Repository owners who want verified fixes, CLI-first developers, and, through the website check, small-business site owners with no code at all, who get a severity-ranked report with "what it means / how to fix" text they can forward.

3. Origins and timeline

Mallace began as **AI-Product-Red-Team-Agent (APRTA)**, a LangGraph prototype that red-teamed AI products. It planned campaigns, modelled threats, generated adversarial payloads (including indirect "trifecta" injections), executed them, judged responses with an LLM, mutated partially successful payloads, and ranked findings with drift detection. On 4 November 2026 the prototype was copied into a new independent repository. On that day it had 47 passing tests and had never run a real campaign. The red-team engine survives in Mallace as the advanced `campaign` command and API. The product's centre of gravity moved to code remediation.

Date	Milestone	Detail
4 Nov	Fork and baseline	Prototype copied, renamed, re-tested (47 pass). Product specification and production roadmap written. Brand palette chosen.
5 Nov	Autonomous security core	50 commits. Policy-bound remediation intents, bounded patch verification, opt-in GitHub delivery, offline secret/Gitleaks/Semgrep/Trivy/Kubernetes/dependency assessment, live AWS S3 configuration scanning, cloud-to-source repair. **First live repair PR** on a private fixture.
6 Nov	Releases a7–a16	Independent confirmation probes, GitHub App authentication, shared per-campaign AI budget ledger, first real budgeted AI campaign (an evaluator miss, see §13), **first public PyPI release (a14)** via OIDC trusted publishing, standalone local runner, hosted identity and one-use runner enrollment.
7 Nov	Outbound runner jobs	The hosted control plane can ask an enrolled customer runner to work without inbound connectivity. Remote find-and-fix live, then `mallace fix` works from any checkout (a23). JavaScript/TypeScript support (a24).
8 Nov	Commercial launch	GitHub Action and hosted `/app` live (a25–a26). **Licensing, accounts, metering and live Stripe (a27)**. Fix prompts moved server-side. License changed to BUSL-1.1 for new releases.
9 Nov	Trust and reach	Duplicate-PR skipping and reviewer recalibration (a28). Free-plan repository proof (a29). Website check, secrets and OSV reports (a30). Hosted website check panel. Marketing site rewritten to describe only what ships.
10 Nov	PHP and Ruby (a31)	Eight new taint rules, Minitest and PHP `assert()` harnesses, pinned Ruby 3.4 / PHP 8.4 images. Live fixture PRs #41, #43, #45, #47.
11–13 Nov	Polish	Website demo and hero work, README rewrite, legal pages final, name cleared.

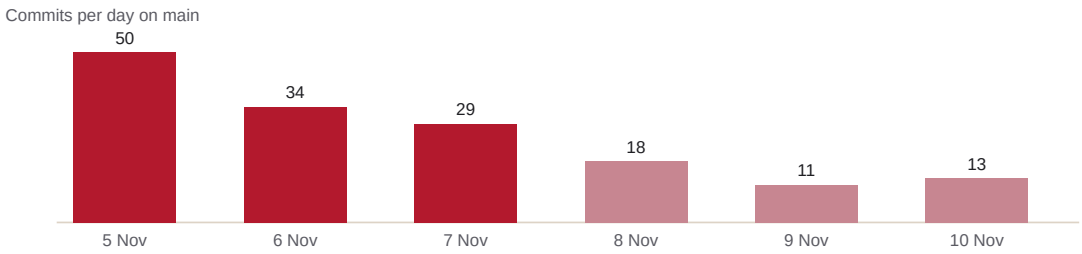


Figure 1. Commit cadence. Breadth-first infrastructure work early in the week; fewer, larger product commits later.

The pivot

The autonomous-security design (5 Nov) set the direction: customers run Mallace against *authorized* repository, AI, cloud and network infrastructure, and it automatically produces fix issues and PRs, with auto-merge opt-in only. The first two days built that general machinery: pluggable assessment adapters, a durable remediation state machine and a customer-controlled runner. From 7 November the work narrowed sharply to the path with the clearest value and the strongest proof story, which was code bugs in GitHub repositories. The broader adapters (S3, Kubernetes, Trivy, the AI campaign engine) remain in the codebase as advanced commands.

4. System architecture

Mallace has three deployable parts and one principle: **customer code stays on the customer's runner.** The only code that leaves is the context around a single finding (capped at 32 files and 128 KB, with common credentials redacted), sent for a single model call and not stored.

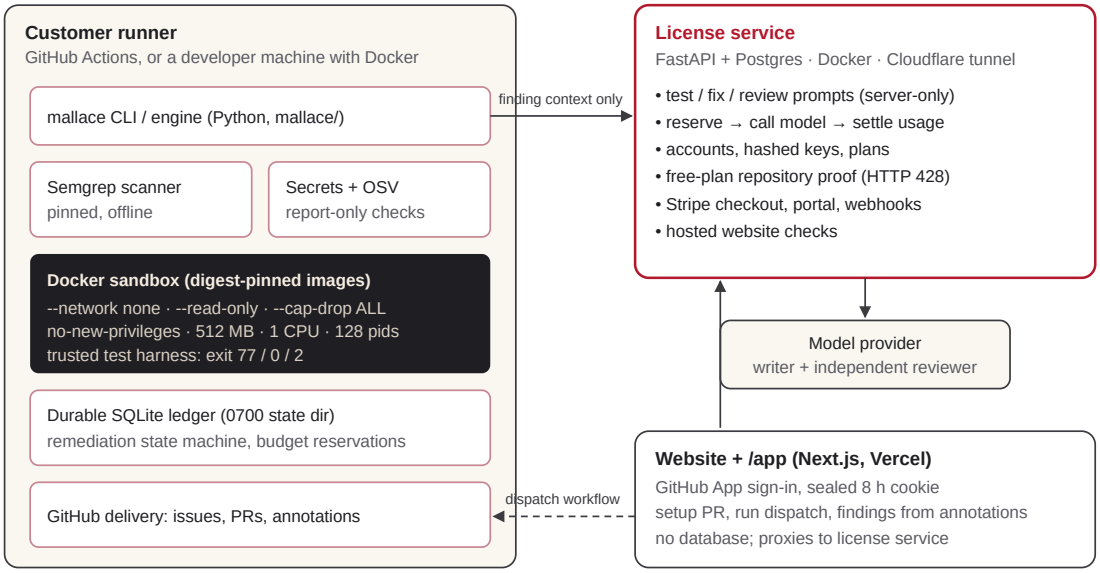


Figure 2. Deployment topology. Scanning, sandboxed tests and GitHub writes happen on the customer runner; the server only brokers model calls, metering and billing.

Code organisation

Area	Contents
`mallace/cli.py`	Unified installed CLI: `fix`, `doctor`, `activate` / `license`, `config`, `site`, `login` / `connection`, `assess` / `targets` / `runner`, `remediate`, `campaign` / `serve` / `secrets`.
`mallace/assessment/`	Scanner adapters (Semgrep, Gitleaks, Trivy, Kubernetes, dependency advisories, AWS S3), SARIF, independent confirmation probes, and the four rule packs (`rules/{python,javascript,php,ruby}.yml`).
`mallace/remediation/`	The fix engine: `autofix.py` (orchestration and language harnesses), `sandbox.py` (isolated container gates), `coordinator.py` / `patches.py` (writer, gates, retest, reviewer), `contracts.py` / `policy.py` (trusted policy), `state.py` / `store.py` (durable state machine), `pr_delivery.py` / `github*.py`, `budget.py`, `doctor.py`, `extra_checks.py`.
`mallace/websites/`	Ownership proof, pinned-IP fetcher and read-only checks for `mallace site`.
`mallace/control/`, `maintenance/`	Hosted identity, runner enrollment, outbound job transport and the local durable worker.
`mallace/graph/`, `agents/`, `taxonomy/`, `trifecta/`, `evolution/`, `scoring/`	The inherited LangGraph AI red-team campaign engine.
`services/license/`	License service. It is never packaged in the wheel, so the prompts stay private.
`site/`	Next.js 16 / React 19 / Tailwind 4 marketing site, docs and `/app`, with Three.js for the hero.

5. The find → prove → fix → review → deliver pipeline

The core loop and the per-language harnesses live in `mallace/remediation/autofix.py` (`_repair_finding`). Container execution lives in `sandbox.py`. Each stage has a single success condition and a set of named failure outcomes, so a run report states exactly why a finding was not fixed.

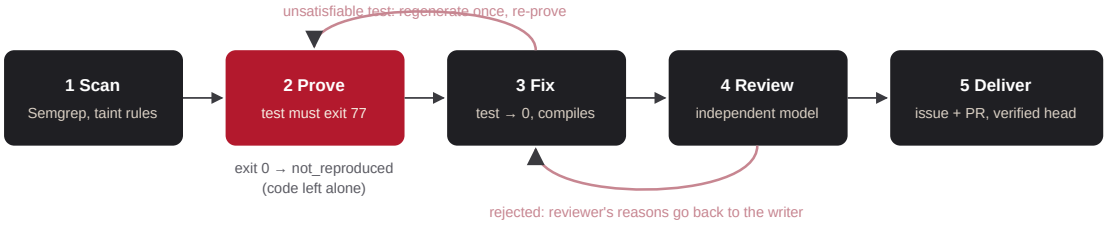


Figure 3. The repair loop with its two bounded feedback edges.

Stage 2 in detail: the proof contract

The model writes only the test. Mallace runs it against the *pinned original revision* inside a trusted, language-specific harness, and the harness, not the model, decides the outcome:

Exit	Meaning	How each harness enforces it
`77`	An assertion failed: the weakness is demonstrated	Python `unittest` failures; Node `node:test` `ERR_ASSERTION`; Ruby `Minitest::Assertion`; PHP `AssertionError`
`0`	All tests passed: not reproduced	The code is left alone (`not_reproduced`) and no repair spend occurs
`2`	Test unusable	Import errors, zero tests, any non-assertion error, all tests skipped. Ruby uses `exit!` to bypass `at_exit` hooks so a test that loads `minitest/autorun` cannot change the result. PHP refuses to run unless `zend.assertions` is on and uses a shutdown guard against fatal errors.

Separating "the bug is shown" (77) from "the test is broken" (2) closes the most common way an AI pipeline fools itself, which is a test that fails for the wrong reason. The test bundle is content-addressed. Its digest is written into the trusted policy, and the same bundle must pass after the patch.

Stage 3–4: writer, gates, retest, reviewer

- The writer may edit only paths the trusted policy allows. Credentials, Git metadata, workflow and policy definitions and protected paths are off limits. **Test changes cannot certify themselves.**
- Gates are administrator-owned argv, never model text: a whole-workspace syntax gate (Python `compile()`, `node --check` plus Node's `stripTypeScriptTypes`, `RubyVM::InstructionSequence.compile`, PHP `token_get_all(TOKEN_PARSE)`) and the original-case retest.
- The retest runs *before* review, so the reviewer judges a patch with known test results. Live run a22 surfaced the opposite order as a defect.
- The reviewer is a separate model call with its own prompt. A rejection's reasons are fed to the next writer attempt. Attempts are bounded.

Stage 5: delivery that survives ambiguity

Intent is persisted before any external write. After an ambiguous GitHub response, Mallace searches for stable markers and reconciles rather than retrying a create. PRs publish only to the mapped repository and base, and the verification binds the exact commit delivered. Each PR carries a plain-English title and explanation (risk, location, how it was proved), the regression test, and the verification facts collapsed underneath. A commit-independent `mallace-fix` key lets later runs skip findings that already have an open PR.

Outcomes are first-class

Every finding ends in exactly one of `fixed`, `not_reproduced`, `test_failed`, `fix_failed` (with a `stage`), or `budget_exhausted`. None of these is rounded up to success.

6. Detection: taint-tracked rules across four languages

Detection uses a **pinned, offline Semgrep** image run with `--pull never` and a versioned Mallace ruleset (currently `semgrep-1.184.0-mallace-code-v6`). The rules use taint tracking, so a finding needs caller-controlled data to reach the sink:

Language	Sources	Sinks
Python	Function arguments (except `self` / `cls`), `sys.argv`, environment, stdin, web `request`	`eval` / `exec`, `os.system` / `os.popen` /shell subprocess, SQL built with `+`, f-string, `%`, `.format`; `yaml.load` without a safe loader and any `pickle` load are always reported
JS / TS	Function parameters, Express `req.query/body/params/headers/cookies`, `process.argv`, `process.env`	`exec` / `execSync` / `spawn(shell:true)`, `eval` / `new Function` / `vm.run*` /string `setTimeout`, SQL built by `+` /template/ `.concat` into `query`, `execute`, `raw`, Prisma `$queryRawUnsafe` …
PHP	Function parameters, superglobals, `$argv`, `getenv`	SQL via PDO, mysqli, pg, `$wpdb`, Laravel `DB::`; `exec` / `shell_exec` / `system` / `passthru` / `popen` / `proc_open` /backticks; `eval`; `unserialize` without `allowed_classes ⇒ false`
Ruby	Method parameters, Rails `params` / `cookies` / `request`, `ARGV`, `ENV`, `gets` /STDIN	`system` / `exec` / `spawn` / `IO.popen` / `Open3.*` /backticks/ `%x()`; SQL string building; `eval` / `instance_eval` / `class_eval` / `module_eval`; `Marshal.load`, unsafe YAML (with `Shellwords.escape`, `to_i` and `Integer()` as sanitizers)

Precision over recall, deliberately

Moving to taint rules in v4 cut a self-scan of Mallace from 12 constant-only SQL false positives to 4 caller-built-identifier cases, while all 6 planted bugs in the fixture were still found. SQL is reported only when a query is *built* from caller data, not when a wrapper passes one through. The proving stage then removes most of what remains: a finding that cannot be shown to fail is never patched.

Making tests able to run real code

- **Third-party dependencies.** A pinned-base image is built from plain `requirements.txt` / `pyproject` requirement lines (no repository code runs), cached and referenced by content ID. `package.json` registry dependencies install with `--ignore-scripts`; Gemfile and Composer dependencies are handled similarly. Tests still run with no network.
- **Module format.** ESM-syntax TypeScript inside a `"type": "commonjs"` package failed to import in the harness. A trusted Node load hook now loads such files as ES modules.
- **Version realism.** On the Ruby fixture, `YAML.load` was correctly *not* reproduced, because Psych 4 made it safe by default. The proof stage did its job and declined to "fix" a non-bug.

7. Safety and trust model

Mallace is a tool that runs AI-written code on customer machines and writes to customer repositories. Its design assumes the model is untrusted, the repository is untrusted, and every external call can fail ambiguously.

Untrusted model output

- The model never chooses scanner invocations, gate commands or the outcome of a test.
- Edits are confined to trusted allowed paths. Test changes cannot certify themselves.
- Invalid generator or judge output fails explicitly instead of defaulting to "safe".

Untrusted repository

- Settings are read from `~/.config/mallace`, never from the repository, so a checked-in file cannot raise spend.
- A profile can only *lower* the budget and cannot change repository or paths.
- Dependency images are built from requirement lines only; repository code never runs during builds.

Containment

- Digest-pinned images; `--network none`, `--read-only`, `--cap-drop ALL`, `no-new-privileges`, non-root UID, 512 MB, 1 CPU, 128 pids, timeouts and output caps.
- The independent reproduction policy is rejected unless its gate is required, network-free and bound to a pinned bundle digest.

Fail-closed accounting

- Budgets default to zero and empty pricing denies calls.
- An upper bound is reserved before each call and settled to actual tokens afterwards. Failed or unknown calls keep their reservation.
- A hard per-run cap ($2 default, $1,000 typo ceiling) covers every paid call in the run.

Data minimisation

Only the context around each finding is sent (at most 32 files, 128 KB), with common credentials redacted. The service stores no code: usage rows hold tokens, cost, model and repository name. Remote-job records on the control server hold only counts, PR numbers and spend, with no paths, code or finding text. This was verified by inspecting the live database and journal after acceptance runs.

Authorisation for anything live

The older assessment and campaign paths require exact approved endpoints, timezone-aware expiry per endpoint, and explicit CIDR lists. Every resolved address must be inside an approved range, and connections pin the literal IP while preserving Host and TLS identity, which defeats DNS rebinding. Requests do not follow redirects or inherit proxies. The AWS S3 adapter reads bucket configuration only, never objects, under explicit account, region, owner and expiry grants. Every acceptance bucket was deleted and its absence verified.

Durability without pretending

The campaign API persists an idempotent queue and a per-case journal. On restart, unfinished work becomes `interrupted` and is never silently replayed. Queued jobs run only if the configuration fingerprint still matches. The documentation states plainly that pending case attempts may have produced target-side effects and that the journal is *not* exactly-once delivery. Runner revocation stopped a live worker in about six seconds. A dropped acknowledgement plus worker restart executed exactly once in live acceptance.

> **Self-maintenance rule.** Mallace can be an enrolled repository under its own policies, but it gets no bypass. It cannot widen its own protected policy or approve its own verification evidence. A daily independent source self-assessment runs in CI from a separately pinned controller.

8. Commercial architecture: licensing, metering and payments

Enforcement by where the code lives

From 0.1.0a27 the client is source-available under the **Business Source License 1.1**, with an additional use grant to scan and to generate fixes with a Mallace key. It converts to Apache 2.0 on 2030-11-08. Earlier releases and the inherited prototype stay MIT. The real enforcement is architectural. The prompts and model calls that write tests, patches and reviews exist only in `services/license/`, which is never packaged, so a patched client still cannot produce fixes without the service.

The license service

Route	Purpose
`GET /v1/account`, `POST /v1/model`, `POST /v1/fixes`	Authenticated with the license key (stored hashed). `/v1/model` reserves an upper bound, calls the model, then settles to actual usage, or marks it failed.
`GET /v1/me`, `POST /v1/me/key`, `/checkout`, `/portal`, `/v1/me/sites`	Authenticated with the user's GitHub token, which the service verifies itself. Used by `/app`.
`POST /v1/stripe/webhook`	Signed and idempotent by event ID. Stripe configures itself on first checkout (product, monthly and yearly prices, webhook, portal).

Plans

Plan	Price	AI allowance / month	Repos with fixes	Websites
Free	$0	$1 (≈3 fixes)	1	1
Pro	$29/mo or $290/yr	$15 (≈50 fixes)	Unlimited	10

Payments use Stripe Managed Payments, with Stripe/Link as merchant of record. A real checkout upgraded a test account to Pro, and a portal cancellation showed the correct "cancelled, ends Dec 9" state.

Closing the free-plan loophole: repository proof

The Free plan allows fixes on one repository, but the repository name comes from the client. 0.1.0a29 added a challenge so the server can check it:

```
# first free-plan model call
client  → POST /v1/model {repo, commit, path, sha256, text}
server  ← 428 Precondition Required
# client retries once with the run's existing GitHub token
client  → POST /v1/model (+ GH token)
server  : read path @ commit from GitHub, compare sha256 / text (credential redactions allowed)
server  ← 200 (proceeds)  or  403 repository_not_verified
```

The token is used for that single read and is never stored or logged. Pro and Owner accounts skip the check and never send the token. The rollout order was itself a lesson: the server half was held back until the public `v0` Action tag pointed at a client that could answer the challenge.

Cost reality

Live runs put actual spend near **$0.07 across three fix runs** during early local acceptance, with reservations of about $0.21–$0.23 per fixed finding against the $2 cap. A single $2 run on the acceptance repository fixed shell, SQL and pickle findings together (PRs #31–#33). That makes the $1 free allowance worth roughly three fixes.

9. The read-only website check

The website check extends Mallace to people with no repository, under a strict rule written into the design document: *scanning a site you don't own is illegal in most places, and an attack can break a production site, so the live-site part only looks, and only after the user proves they own the site.*

Design choice

Three approaches were considered: wrapping OWASP ZAP baseline or Nuclei in containers (broad, but heavy, noisy and partly intrusive), a hosted scanner (which turns the server into an open internet scanner), or a small built-in Python checker using the existing `httpx` dependency. The third was chosen. Every request is one a normal browser could make, and the findings are written for non-experts. The same module later powered the hosted "Your websites" panel.

Ownership

A per-site token, placed as a DNS TXT record (looked up over Cloudflare DNS-over-HTTPS), a `/.well-known/mallace-verification.txt` file or a meta tag. Ownership is **re-proven at the start of every check**, so access ends when the proof is removed.

Request discipline

GET/HEAD only, at most 30 requests (about 20 in practice), one at a time with pauses, 10 s timeout, 256 KB read cap, self-identifying user agent. Private, loopback and link-local addresses are refused, and connections go only to the vetted IP. Redirects must stay on the same site.

Checks: HTTPS and certificate validity, expiry within 14 days, HTTP→HTTPS redirect, HSTS, CSP, clickjacking protection, `nosniff`, cookie `Secure` / `HttpOnly` / `SameSite`, risky CORS, mixed content, version disclosure, exposed `.env` / `.git` / `.svn` / `.htpasswd` /SQL dumps/backups/ `phpinfo` / `server-status`, directory listing, outdated WordPress and `security.txt`. Exposed-file findings require content that matches the file type, which avoids soft-404 false alarms.

Where it runs: from the CLI it runs on the user's machine and sends nothing to Mallace. From the app it runs on the license server, with a 120 s cooldown per site. The hosted path is deployed but, per the status notes, had not yet been exercised end to end by a signed-in user.

10. Release engineering and evidence discipline

Pipeline

Three GitHub workflows: `checks.yml` (product tests on a Python 3.12 and 3.15 matrix, installed-wheel checks, operator JavaScript checks), `source-assessment.yml` (Mallace scanning itself with an independently pinned controller), and `publish.yml`.

- **Hash-bound promotion.** A wheel is promotable only if two successful CI runs (product and source) exist for the exact commit, and its source, hash and version match.
- **Validate-only by default.** `publish.yml` performs a real isolated pipx install of the candidate and skips upload unless publishing is explicitly requested.
- **OIDC trusted publishing** to PyPI from a separate wheel-only job, restricted to `main`, with attestations. No long-lived tokens exist.
- **Install acceptance** uses both pip and uv in isolated caches. A shared uv cache once returned the previous alpha; that was recorded as a failed check and fixed with per-workspace cache isolation rather than a global cache wipe.
- **Server rollout** installs versioned releases from a hash-verified wheel, keeps the prior release for rollback, and finishes with an authenticated real job plus checks that anonymous requests get 401.

Evidence as a deliverable

`docs/evidence/` holds dated JSON and text receipts: CI run IDs, wheel SHA-256 hashes, PR numbers, reservations, database inspections and container smoke tests. The implementation journal records each rollout with the same identifiers. Claims in the README and on the marketing site are limited to what these receipts support. The site was rewritten on 9 November to "describe only the shipping product", and a standing rule now enforces that.

Development model

Mallace was built by one owner directing AI coding agents, with explicit role separation recorded in the repository: one agent owns the frontend (`site/`), backend agents own scanning, repair, APIs, packaging and operations. Plans live in `docs/superpowers/plans/` and specs in `docs/superpowers/specs/`, with test-first red/green changes and independent reviews before deployment. Deployment and publishing remain owner-run steps. The agents prepare commands, and the owner dispatches production cutover and PyPI publication.

> **Why this matters.** Shipping 31 releases in a week is only safe if each one is reproducible and attributable. The combination of exact-commit CI pairs, hash verification, retained previous releases and recorded receipts let the project move quickly without losing track of what is actually deployed.

11. Live acceptance results

All live runs used repositories, buckets and sites owned by the project. Fixture acceptance is real execution against real infrastructure, but it is not customer production acceptance.

Release	Scenario	Result
early	Local end-to-end, real Semgrep, writer + reviewer models, Docker, GitHub	PRs #4 (pickle → JSON sessions), #6 (parameterised SQL), #8 (argument-list `tar`). The first `backup.py` run was reported `fix_failed` after reviewer rejections, then fixed once reviewer reasons reached the writer.
a22	Remote job: hosted service asks a paired runner	`fixed: 1`, PR #10 (f-string SQL), $0.22 reserved. The server stored counts, PR number and spend only.
a23	Dependency-needing Python bug	`yaml.load(..., Loader=yaml.Loader)` needing PyYAML proven and fixed to `safe_load`, PR #12, $0.21.
a24	JavaScript / TypeScript	CommonJS `exec('tar ' + dir)` → `execFile('tar', [..., '--', dir])`, PR #14 ($0.22). Remote TS template-literal `LIKE` query, PR #16 ($0.23).
a26	Public Action on GitHub-hosted runners	Scan found 8 findings, all read back by the app. A $2 fix run gave PR #20 (SQL), PR #22 (pickle), one `fix_failed`, 5 skipped by `max-findings`, ≤$0.87 reserved. PR #24 (unsafe YAML) with the new plain-English body.
a27	Through the metered license service	Live fix opened PR #25. A real Stripe checkout and portal cancellation behaved correctly.
a28	Reviewer recalibration	Shell, SQL and pickle findings all fixed in one $2 run, PRs #31–#33.
a31	PHP and Ruby	Ruby command and SQL injection (PRs #41, #43), PHP command and SQL injection (PRs #45, #47). Ruby `YAML.load` correctly not reproduced.
a5–a7	Cloud and confirmation	Real S3 configuration scans on disposable buckets. S3 → CloudFormation source repair PR with passing CI. Independent probes confirmed eval and SQL positives, rejected a constant-eval alert and kept parameterised SQL clear.
a16–a18	Hosted identity and runner transport	GitHub device-flow login, one-use enrollment, replay rejection, revocation (≈6 s stop), dropped-ack exactly-once execution. No client secrets in the server DB or journal.

12. Defects found by running it for real

Almost every live run surfaced a defect that unit tests had missed. Each was fixed with a regression test and recorded. This is the strongest argument in the project for doing paid, bounded live runs early.

Symptom in a live run	Root cause and fix
Reviewer rejected both `backup.py` patches; finding reported `fix_failed`	Reviewer reasons were not reaching the writer. Feedback is now threaded into the next attempt.
Reviewer judged patches without test results	Retest now runs before review.
Model-written test asserted something impossible	An unsatisfiable test is detected, regenerated once and re-proven (`generation` loop of two).
Remote jobs couldn't find a mise-installed `gh`	The absolute `gh` path is pinned at registration.
Default ledger refused to open	A permissive pre-existing state directory. It is now tightened to 0700.
Provider settings read before key and models were set	The run re-executes in an isolated child process.
Fresh machine failed every scan	Scans use `--pull never` . `doctor` and `fix` now pull missing digest-pinned images automatically.
First JS attempt ended `test_failed`	The prompt asked for both `child_process` and `node:child_process` mocks, and Node 24 treats them as one module. The prompt now asks for one.
Scan path outside the fix profile registered silently, attempted nothing	Registration now fails and names the paths.
ESM TypeScript in a CommonJS package wouldn't import	A trusted load hook loads such files as ES modules.
Correct `os.system` → argv fix rejected over exit-status details	Server-side reviewer prompt now accepts standard safe idioms (a28).
PRs showed only digests and jargon, without the test	PRs now carry a plain-English explanation plus the regression test (a26).
First budgeted campaign failed before any dispatch	AutoGen timeout rejected by an allowlist, which the preflight had masked by replacing factories. Fixed, and the preflight now intercepts only the HTTP transport.
Evaluation accepted redacted evidence as HARDENED	Transport redaction flags were ignored by the evaluator. Redacted evidence now stays `UNKNOWN` before any provider is constructed.

13. The honest miss: the first budgeted AI campaign

On 6 November the inherited red-team engine ran its first real campaign against an owned target under a $0.50 grant: six model requests, three target requests and $0.254301 reserved. The target's independent oracle recorded a synthetic **canary leak**, but the LLM judge classified every case `HARDENED` and the campaign reported zero findings.

The project recorded this as **a real evaluator miss, not passing acceptance**. The original failed record was kept immutable, the grant was treated as consumed, and no retry was assumed. A separately approved one-request recheck (a $0.10 cap, $0.060860 reserved, zero target requests) gave the judge corrected owner context and a static system rubric, and it classified the saved leak `VULNERABLE`. Because the original payload was unavailable, this was documented as a saved-case reference evaluation, not exact replay or general accuracy validation.

Investigating the miss uncovered a second, more serious bug. Redaction and scrubbing flags were being ignored by evaluation, so altered evidence could be judged `HARDENED`. Four regression tests reproduced it before the fix.

> **Why it belongs in a case study.** This episode is the clearest illustration of the product's design philosophy. An LLM judge is a probabilistic classifier, and its "safe" verdict is not evidence. It is also the reason the code-fix product uses an executable proof (a failing test in a sandbox) instead of a model's opinion to decide whether a bug is real.

14. By the numbers

155
commits on main (5–13 Nov)

16.8k
lines of Python in `mallace/` (150 files)

14.2k
lines of Python tests (147 files)

6.3k
lines in site app + lib (Next.js)

1,174
unit tests (at a27)

43 + 30 + 9
integration · site · license-server (real Postgres)

17
prove-and-fix rules, 4 languages

20+
verified fixture fix PRs opened (#4 → #47)

Line counts are from `git ls-files` on 14 Nov 2026. Test counts are the totals recorded in Project Status at 0.1.0a27. The test-to-code ratio is close to 0.85:1.

Technology stack

Layer	Choices
Engine	Python ≥3.12 (validated on 3.12.14 and 3.15.2), Pydantic contracts, SQLite ledgers, httpx, LangGraph/AutoGen (campaign engine)
Analysis	Semgrep 1.184 (pinned image, custom taint rules), Gitleaks, Trivy, OSV API
Sandbox	Docker with digest-pinned Python, Node 24, Ruby 3.4 and PHP 8.4 images
Service	FastAPI, Postgres, Docker Compose, Cloudflare tunnel, Stripe Managed Payments
Web	Next.js 16, React 19, Tailwind 4, Three.js, Vercel Git deployment
Delivery	PyPI (OIDC trusted publishing and attestations), composite GitHub Action, GitHub App

15. Limitations and what comes next

Known limits (stated in the product's own docs)

- **Narrow coverage.** No Go, Java, Rust or C#. No XSS, SSRF, authentication or business-logic bugs. Secrets and dependencies are reported, not fixed.
- **Not a penetration test.** The website check looks but doesn't attack.
- **Single-box backend.** The license service runs on one server. An outage stops fixes and in-app website checks (scanning still works).
- **GitHub-only identity.** No email sign-in. Customer installation onboarding and tenant identity are partly implemented.
- **Unverified paths.** The hosted website check has not been run end to end by a signed-in user. The AI campaign engine has no reviewed evaluation benchmark, and the shipped API container defaults to simulation for campaigns.
- **Operational gates.** Production monitoring, retention and off-host restore drills, distributed scheduling and safe graph resume remain open.

Next: sandbox attack testing (planned, not started)

Phase 3 of the website-check design builds the application from the repository inside Mallace's sandbox and actively attacks that copy (SQL injection, XSS, login bypass, cross-user access), then fixes the code through the existing PR flow. The real site is never touched. This would extend the "prove it before you fix it" principle from unit-level tests to running applications, and it would open the bug classes that static taint rules handle poorly.

16. Lessons learned

1. **Make the oracle executable.** A failing test in a locked-down sandbox is a better arbiter than any model's judgment. The one place the project relied on an LLM verdict (the campaign judge) produced its most significant miss.
2. **Separate "broken test" from "bug shown".** The 77/0/2 contract, enforced identically across four languages, is a small idea that removes a whole class of false confidence.
3. **Run it for real, early, on a budget.** Over a dozen defects appeared only in live runs, and most cost under $0.25 to find. Hard caps and reservation accounting made it safe to do this often.
4. **Put enforcement where the code can't be edited.** Moving prompts server-side made licensing robust without obfuscating the client.
5. **Fail closed and say why.** Named outcomes (`not_reproduced`, `fix_failed:stage`, `budget_exhausted`) turn a non-fix into useful information rather than silent failure.
6. **Narrow to win.** Two days of general infrastructure (cloud, Kubernetes, AI campaigns) paid off only after the product narrowed to one clearly valuable path. The general machinery (policy, state machine, delivery) carried over unchanged.
7. **Evidence over adjectives.** Receipts with run IDs and hashes, plus a rule that public copy may describe only what ships, kept a very fast project honest about its own state.

> **In one sentence:** Mallace shows that AI-written security fixes can be trustworthy when the AI is never asked to be the judge. It writes the test, the sandbox decides, a second model reviews, and a human merges.

Engineering case study by CodeFront Labs, prepared 14 November 2026 from the Mallace repository, design notes and recorded acceptance evidence.