OpenFlame

Everything the platform does: building software from a description, delivering it, hosting it in your own cloud, operating it, managing its cost, keeping it running through a disaster, extending it with ready-made services, and letting AI agents drive all of it.

Product overview · Engineering platform and service store

What OpenFlame is

OpenFlame turns a description of a product into a complete, running software system, and then gives the team everything needed to keep building, shipping and operating it. The system runs on Kubernetes clusters in the customer's own cloud account, in a repository the customer owns, on open-source components. Nothing about it requires staying on OpenFlame.

A person describes what they want, in words, files or both. OpenFlame plans the system, generates it, reviews its own work, opens a pull request and deploys the result: backend services, a database, a GraphQL API, sign-in, web and mobile apps, and the pipeline that builds and ships them. From then on the same console is where developers work, releases are promoted, production is watched and the cluster is managed.

In this document

1 **Create and build**
From a description to a reviewed, working system; the workspace; debugging; AI review; architecture checks

2 **Test and deliver**
The pipeline, previews, promotion, feature flags, test data, load testing, accessibility, upgrades

3 **Host**
Clusters in your own cloud, the platform stack, networking, domains, mail, databases

4 **Operate, monitor and manage**
The console: clusters, services, logs, metrics, alerts, errors, on-call, status page, analytics

5 **Secure**
Supply chain, cluster isolation, secrets, vulnerability monitoring, compliance, GDPR, data residency

6 **Cost management**
What each cluster and service costs, and what the same workload would cost on another cloud

7 **Business continuity**
Disaster recovery across clouds: sync, standby, failover, drills

8 **The service store**
Eighteen ready-made services and two cluster add-ons an application can install

9 **Built for agents**
Everything above, available to AI agents through MCP, with limits a person sets

10 **Ownership and no lock-in**
What the customer owns, and what leaving looks like

Each project also has a Business area, in which a team of AI agents helps run the business. It is described in a separate document.

1 Create and build

A project starts from a description and arrives as a pull request containing a whole system, already reviewed. After that, developers keep working on it in browser workspaces with AI agents alongside.

Starting a project

- **One "New project" form.** Leave the prompt empty for a plain project, or describe the product, attach files, or both. Text, Markdown, data files, source code and images are accepted, and files alone are enough to launch.
- **A guided launch on one page.** A stepper takes the project through choosing a cloud and cluster size, creating the development cluster, installing the platform, creating the developer's workspace, linking AI accounts, choosing a model, planning, building, review and merge. Each step says what it is waiting for.
- **It picks up where it left off.** Completed steps are recorded, so the launch reopens at the right step on any device. If something is undone later, such as a deleted cluster, the page flags it and offers to go back to that step.
- **Launch later, or onto an existing cluster.** A project created without a prompt can be launched from its overview, and uses the development cluster it already has.
- **Match an existing website.** Name a reference site and OpenFlame reads it, after confirmation and respecting the site's crawling rules, and hands its pages and chosen assets to the planner and the screen builder.

What a project receives

- **A full application stack.** Postgres, a GraphQL API (Hasura), sign-in (Ory Kratos, with Ory Oathkeeper at the edge), a cache (Valkey), a message bus (NATS), object storage (RustFS), a service mesh (Istio), GitOps deployment (Argo CD) and monitoring (Grafana, Loki, Tempo, Mimir).
- **Backend services in six languages.** Go, Python, Node, Java, C# and Rust. Each has a gRPC template with a REST gateway and a plain HTTP template: twelve templates in all.
- **Sign-in that is already wired.** Email and password, social providers such as Google, extra profile fields, and session rules applied at the edge, so every service knows who is calling.
- **Web and mobile apps.** React web apps and Expo (React Native) mobile apps, with sign-in pages, typed GraphQL and generated API clients in place.
- **One readable repository.** Data, backend, frontend, infrastructure, integrations and brand, plus a context folder holding the brief, specifications, decisions and architecture notes.
- **API contracts kept in sync.** Five sources define each API: the service's interface definition, the services record, the public routes, the data schemas and the migrations. OpenFlame generates the API reference, client code and access rules from them, and the pipeline refuses a service that was not registered.
- **A guide for coding agents.** Every project carries an agent guide and a set of skills, kept current by OpenFlame through pull requests.

The architect: from description to pull request

- **Plan first, then build.** An agent turns the description into a structured plan: data models, APIs, services and their languages, apps, sign-in methods, integrations and environments. The plan is shown as cards that can be edited before anything is built.
- **A build in clear stages.** Generation runs in the developer's own workspace on its own branch: data models and migrations, APIs and services, app shells and clients, sign-in, integrations, the pipeline, then each screen one at a time with checks after each, then the features working together, then review. Every stage is a commit, so an interrupted run continues where it stopped.
- **Ready-made services in the plan.** Where a store service fits, the planner chooses it over generating new code, and says why.
- **No clock limits on agents.** Long agent turns are watched for stalls and resumed in the same session. The chosen model is never swapped silently.
- **Request changes after the build.** Describe a change on the launch page, with files if needed, and the agent applies it to the open pull request, re-runs the checks and pushes.
- **A list of what still needs setting up.** After a build, the launch page lists each outside account the system needs: which key goes where, webhook addresses, and redirect addresses for sign-in providers.

Agents, models and AI accounts

- **Bring your own AI subscriptions.** Each developer links their own AI accounts in the console: Anthropic, OpenAI, Google, Moonshot, xAI, Devin and Meta. A proxy inside the workspace serves them to the agents, and the accounts never leave the workspace.
- **Choose the model.** Any coding model the linked accounts offer can be the default, or be chosen per plan, per conversation or per reviewer.
- **Coding agents in every workspace.** Claude Code, Codex, Cursor CLI and opencode are installed. The architect, reviews and fixes run on opencode, and the others are available in the terminal.
- **Tools that update safely.** Agent tools are installed from a signed manifest, with stable and beta channels, and can be updated from the console without rebuilding the workspace, never while a build is running.
- **Vibe: change the app by conversation.** Each conversation works on its own branch. The diff is visible, and one click opens a pull request. Several conversations can run side by side.

The developer workspace

- **A browser IDE per developer.** Each developer gets a workspace on a development cluster: VS Code in the browser with all six language toolchains, protocol tools, Kubernetes and GitHub command-line tools, a browser for testing and a throwaway Postgres.
- **Only the owner gets in.** A workspace's editor and API admit only that workspace's owner, and the mesh blocks other workloads from reaching it.
- **Always current with main.** The workspace keeps its checkout up to date. The Explore pages read what is on main and badge each item "On main", "In a pull request" or "Local only".

- **Browse what exists.** Data models with diagrams, fields, relations and migrations; API reference pages with "Try it"; the sign-in configuration; and a map of services.
- **A workspace API.** The console and agents use it to scaffold services, apps, schemas, routes and pipelines, and to run plans and reviews.

Running and debugging your version

- **Debuggers for all six languages.** From the editor, "Run and Attach" starts or finds the service and attaches the right debugger: Delve for Go, the JavaScript debugger for Node, debugpy for Python, the Java debugger, netcoredbg for C# and CodeLLDB for Rust.
- **Start a service from the console.** The Services page starts any service in a developer's workspace and shows where each service runs and what resources it uses.
- **Live traffic to your version, with no pull request.** A developer's in-progress version answers on its own address, or the shared API sends only that developer's requests to it. Everyone else keeps using the deployed version, and the routing is removed when the developer stops.

AI review

- **Review before the pull request opens.** The project's checks run, then fresh reviewer sessions that never saw the builder's reasoning read the change through three lenses. Each finding must survive a verifier that tries to disprove it before it is fixed, and each round's fixes are committed only when checks pass.
- **Two levels.** Standard fixes serious findings and above in up to four rounds, and lists the minor ones. Ultra adds three reviewers per lens, majority verification, a completeness pass and repeated test runs, and fixes down to minor findings in up to twelve rounds.
- **Several models, checked by a different one.** The team picks which models review, which verify and which fix. By default a finding is confirmed by a different model from the one that raised it.
- **Reviews that run the project as the pipeline does.** A review repeats tests to catch flaky ones, then waits for the real pipeline on the pushed commit and treats a failure as a finding.
- **Watch it and stop it.** The round, the phase and each finding's state are shown, with "Stop after this round" and "Stop now". A review pauses when AI allowance runs out and resumes by itself.
- **Pull requests driven to mergeable.** On an open pull request, a loop feeds outside reviewers' comments and failing checks to a fix turn, pushes, replies and resolves threads until every check passes, then offers Merge.
- **Review any pull request.** A review at either level can be started from the CI screen, by a label, or automatically for every pull request.
- **Design review of every interface.** Started from any app, it builds the app and captures every screen at phone, tablet and desktop widths, in light and dark, in each language including right-to-left ones, and in its loading, error and empty states. It measures layout and contrast, reviews through six lenses against the brand, and re-captures to prove each fix.

Architecture that stays true

- **A check on every pull request.** OpenFlame works out how the system's parts connect before and after a change and reports the difference in plain language, such as one service starting to read a table another service owns.
- **Rules a project can enforce.** No outward call outside the gateway, no service reading another's tables, no circular dependencies. A recorded baseline means an existing project is not blocked on day one.
- **Decisions on record.** Design decisions are kept as files in the repository, each with a checkable rule where possible. A change that alters the architecture without one is asked for it, with a draft offered.
- **Two ways to resolve a conflict.** When a change breaks a decision, OpenFlame offers either a code change that honours it or a new decision record that replaces it. A person chooses.
- **Documentation that keeps up.** The services map, data-model diagrams, API reference and architecture notes are regenerated by pull request when they drift. A person's own writing is never overwritten.
- **"How this system works".** An always-current page, readable in every environment, with a history of how the architecture has changed.

Mobile development

- **Installable development builds.** The cluster builds an Android development app that installs from a QR code. Store builds for iPhone and Android go through Expo's build service.
- **A QR code for every version.** One code opens a developer's live version from their workspace. Each environment hosts its own app bundle with its own code, and so does each pull-request preview. A build of the wrong native version is refused with an explanation.

2 Test and deliver

Every project gets a complete delivery pipeline without writing a pipeline file. It runs inside the project's own cluster, on Argo Workflows and Argo CD, and the project's repository is the record of what runs where.

On every pull request and merge

- **No pipeline files.** A push or a pull request starts a run. The console chooses which cluster builds and which hosts previews.
- **Builds for every component.** Services, web apps and mobile apps are built as container images from each component's template.
- **Tests and checks.** Type checks, lint, unit tests and web builds for apps, and unit tests for Go, Node and Python services. A check that could not run is never reported as passed.
- **A structure check first.** Before building, every service is checked against its template and registration, with the file and the fix named.
- **Only what changed.** A push rebuilds the components it touched, plus any whose deployed version is behind. Within a pull request, a new push builds and tests only what changed since the last passing run, and the check lists what was built and what was reused.
- **Newer commits stop older runs.** On a branch, a newer commit stops the older run and marks it "Superseded". Runs on main queue in order.
- **Nothing gets lost.** A commit on main that started no run, for example because a notification was lost, gets one within five minutes. Very large pushes build too.
- **A failure report that names the problem.** One check per run on the pull request lists the failed steps, the failing tests by name and the relevant log lines, with a link to the console.
- **Fix with AI.** On a failed run, OpenFlame reproduces the failing tests, diagnoses the cause, proposes a change and verifies it. It warns when a fix would only hide the problem, and applies nothing until a person agrees.
- **Notifications.** Every finished run appears in the console's feed and can go to Slack and email, per project and by kind of result.
- **Re-run from GitHub.** "Re-run" on the check starts a fresh run of that commit.

Preview environments

- **A preview per pull request.** Each pull request gets its own copy of its services and interfaces on a development cluster, with a link and a QR code for mobile.
- **Or a reason why not.** If a preview cannot be created, the pull request gets a one-sentence explanation. Stale previews are cleared automatically.
- **A safe database for schema changes.** A pull request that changes the database gets its own copy, restored from backup.
- **Never on staging or production.**

Deploying

- **GitOps deployment.** After tests pass on main, the new versions are written into the repository and Argo CD rolls them out.
- **Deploy guards.** The deploy step writes only the entries it owns and refuses any change that would drop hand-written settings.
- **Confirmation that it is live.** The run waits until the cluster is healthy on what was merged. It fails only when progress stops, never on a fixed clock.
- **Progressive delivery.** Canary or blue-green rollouts with Argo Rollouts, with promote, abort and retry in the console.
- **Database changes in the pipeline.** Numbered migrations and GraphQL settings are applied before deploy and on promotion. An existing database can be adopted with a baseline.

Environments, promotion and rollback

- **Dev, staging and production.** Each environment has a release manifest in the repository recording every component's version.
- **Promotion plans with contract checks.** Before promoting, seven dependency rules check that the API routes, GraphQL fields, migrations and mobile versions each component needs are present in the target. Anything that would break is a blocker.
- **Promotion as a pull request.** A promotion opens a pull request on the target environment's manifest. The checks are repeated there, the cluster applies it on merge, and any application secrets the target lacks are copied over.
- **A releases screen.** A grid of which version runs in each environment, the promotion planner, history, rollouts and a dependency graph.
- **Rollback.** One action promotes the previous versions back. Database rollbacks that would lose data are blocked.
- **Store services promote too,** by their pinned version.

Mobile delivery

- **Bundles hosted per environment.** Each environment serves its own app bundle, so a development build can open any environment's version without a third-party host.
- **Over-the-air updates.** Each environment has its own update channel.
- **Store builds only when needed.** A native build starts only when the app's native parts have changed. Signing stays with Expo, and the platform needs only a token.

Feature flags

- **Release separately from deploying.** A feature can be turned on for particular users, groups or a percentage of people, and changed per environment from the console without a deploy.
- **Declared in the repository.** Each flag has a name, a type, a default, an owner and an expiry.

- **Fast and always available.** Flags are evaluated locally in services, web apps and mobile apps, in all six languages, through an open standard interface. They keep working when nothing else is reachable.
- **Kill switches and gradual rollouts.** A percentage rollout stays consistent for each user, and a feature can be switched off at once.
- **Controlled and recorded.** Production flag changes are limited to owners, and every change is in the audit record and shown beside releases and incidents.
- **No stale flags.** A standing check finds flags that are fully rolled out, and an agent offers a pull request removing them from the code.
- **Experiments.** A flag can run an A/B test, with results analysed in the product analytics tool.

Realistic test data

- **Synthetic data from the data model.** Realistic shapes and volumes that respect relations and constraints, repeatable from a seed, with scenarios a project can define.
- **Masked copies of production.** Personal fields are replaced, hashed or dropped by rules per field. Masking happens on the production side, so unmasked data never leaves it, and a scan fails the copy if anything personal survives.
- **Staging cannot reach real people.** Email addresses, phone numbers, push tokens and payment identifiers are neutralised, and sign-in accounts are replaced with test accounts.
- **Controlled.** Only owners can run a masked copy, every run is recorded, and previews and staging choose which dataset they use.

Load and performance testing

- **Scenarios from real use.** Load scenarios are generated from the project's APIs and its real traffic mix, and can be edited.
- **Safe by design.** Tests run against a preview or staging with synthetic data, with outside integrations in test mode. The cost of a run is shown before it starts.
- **Honest results.** Throughput, response times, error rate, how hard each component was working, and the first thing that broke.
- **A performance check per release.** An optional gate compares each release with a recorded baseline and flags a slowdown.
- **Capacity answers.** Roughly how many users a given size of cluster supports, tied to the cost and right-sizing figures.
- **Fix with AI.** A slowdown can be handed to an agent to trace to the slow query or code path.

Accessibility and language checks

- **Accessibility on every change.** Each pull request that changes an interface is checked, with screenshots. Only new problems fail, and each can be handed to "Fix with AI".
- **Every string in every language.** The project declares its languages once. A check confirms all user-facing text is translatable and present in each language.

- **Translation in the pull request.** New or changed text is translated by an agent using the product's glossary and tone, marked as machine-translated until a person approves, and never replaces a human translation.
- **Layout verified.** Right-to-left layouts and text that grows in translation are checked by capturing the screens.
- **Beyond the app.** Emails, notifications and text messages are covered too, with a page showing coverage per language.

Upgrades done for you

- **What is behind, and what matters.** An inventory of each project's languages, frameworks and tools with their support end dates, ranked by urgency.
- **The upgrade carried out.** An agent follows the official migration notes, fixes what breaks, runs the full checks and the AI review, and for interface frameworks captures every screen to prove nothing changed.
- **One reviewed pull request per upgrade.** It says what changed, what was verified and what could not be. It is never merged automatically, and tests are never weakened to make it pass.
- **Safe to try.** Each upgrade gets a preview, and runtime and database upgrades roll out gradually.
- **Template improvements.** A project generated earlier can take improvements made to the templates since.

Releasing the platform itself

- **Versioned releases.** Every platform release has its own version number and is never overwritten. Publishing a release changes no cluster.
- **Each cluster names its version.** A one-line pull request moves a cluster to a new platform version, and reverting it moves the cluster back.

3 Host

Every project runs on Kubernetes clusters in the customer's own cloud account. OpenFlame creates them, installs the platform, upgrades it and manages it. The clusters and the data stay the customer's.

Clusters in your own cloud

- **Create a cluster from the console.** Pick the environment, cloud, region, Kubernetes version, node size and node range. Versions and sizes come live from the cloud.
- **Seven clouds.** Google Cloud, AWS, Azure, DigitalOcean, Oracle Cloud, Scaleway and Vultr.
- **Advanced options.** Automatic upgrades, a maintenance window, spot nodes, disk size, zonal or regional placement, labels and taints, each in the cloud's own terms.
- **Connect a cluster you already have.** One install command joins an existing cluster.
- **Credentials held safely.** Each organisation's cloud credentials are checked against the cloud before saving, stored in its own secret store, and never shown again. On AWS, a role that trusts OpenFlame can be used so that no key is stored at all, with a least-privilege policy ready-made.
- **Node autoscaling.** Clusters scale their nodes between the chosen minimum and maximum. On AWS the autoscaler is confined to that cluster's own node groups by a permissions boundary.
- **Clean removal.** Deleting a cluster, after a typed confirmation, removes its cloud resources and DNS records.
- **Cluster roles.** Development clusters also host the pipeline, previews and developer workspaces. Staging and production clusters run only the application and change only through promotion.
- **Managed database option.** Staging and production can use the cloud's managed Postgres beside the cluster, with no application change.

How the control plane reaches a cluster

- **The cluster dials out.** An agent on each cluster connects outwards and collects its commands. Nothing connects into a customer's cluster, and every console view of a cluster travels over this channel.
- **Signed commands, checked against the customer's policy.** Each command is signed, expires, and says who asked. The agent checks it against a policy file in the project's own repository and records it in a tamper-evident log.
- **A least-privilege agent.** After setup the agent cannot install cluster-wide components or alter its own policy or records.

What is installed on every cluster

COMPONENT	WHAT IT IS FOR
Istio	Service mesh: encrypted identity between services, traffic policy and the gateway. New clusters use its sidecar-free ambient mode.

Cloudflare Tunnel	All web traffic in, over an outbound connection, with no public load balancer.
CloudNativePG	Postgres 18 with continuous backup, clones and restores.
Hasura	The GraphQL API over the database.
Ory Kratos and Oathkeeper	User sign-in, and the access proxy that checks every request.
NATS JetStream	The message bus for services and platform events.
RustFS	Object storage for files, backups, logs and build output.
Valkey	Cache, with automatic failover.
Infisical and External Secrets	Secrets, delivered from the secret store into the cluster.
Argo CD	Deployment from the project's repository.
Argo Workflows and Argo Events	The build pipeline, promotions and scheduled jobs.
Argo Rollouts	Canary and blue-green releases.
Grafana, Alloy, Loki, Mimir, Tempo, Alertmanager	Dashboards, logs, metrics, traces and alerts.
Haraka	Outgoing and incoming mail, with mail capture on development clusters.
Push gateway	Mobile push notifications.
Webhook validator	Checks that each inbound webhook really came from its sender.
OpenFlame agent	The control channel and the gateway for outside integrations.

- **Pinned versions.** Every component has a pinned version, which can be overridden per cluster.
- **A registry.** A container registry holds platform images, each project's images and build caches, with separate accounts per cluster.
- **A mirror of third-party software.** Third-party images and charts come from OpenFlame's own mirror, pinned by digest and signed, so a cluster does not depend on upstream hosts staying available.
- **Staged mesh upgrades.** The mesh moves one version at a time, cluster by cluster, with a way back at each step.
- **Model serving.** Clusters can serve AI models themselves, with models kept in the cluster's object storage.

Networking and domains

- **A built-in address for every cluster.** Each cluster gets its own domain with simple one-level names for the web app, API, sign-in, GraphQL, admin tools and previews. The tunnel, DNS records and certificates are created automatically.
- **Custom domains in the customer's own Cloudflare account.** Each custom domain gets its own tunnel, created with the customer's token in the customer's account, and its own sign-in instance. OpenFlame's account is never involved.
- **Switch a domain between environments.** A domain can be moved, say from staging to production, with checks before and after and one action to switch back.
- **Direct load balancers for non-web traffic.** File transfer and call media get their own load balancer, open only on their ports and only to listed address ranges.

Mail

- **Sending from the project's own domain.** Per environment, application mail goes through the customer's own mail-sending account or any SMTP provider.
- **Development mail.** A development cluster can capture mail into a console inbox, so sign-up and recovery can be tested, or send from its built-in domain through the platform's relay without holding any credential.
- **Inbound mail.** Incoming messages are delivered, signed, to the cluster over the tunnel and reach services on the message bus.

Databases and storage

- **Continuous backups.** Every database archives continuously and takes a daily backup, kept for 14 days.
- **Backups wherever you choose.** The cluster's own object storage, Amazon S3 or any compatible service, Google Cloud Storage or Azure Blob Storage.
- **Restore to any moment.** A restore creates a new database from the latest backup or from any point in the recovery window.
- **Restore tests.** Production restores its latest backup into a throwaway database every week and alerts if that fails. Production runs two database instances.
- **Storage per purpose.** Files, backups, monitoring data, models and build logs can each use the cluster's store or an outside one.

Cluster add-ons

- **File transfer server.** SFTP for partners who need it, stored in the project's object storage.
- **Media server.** Carries the audio and video for calls and voice agents.

4 Operate, monitor and manage

The console is where people run what OpenFlame builds. Inside a project, the Engineering area groups its pages as Develop, Operate, Observe and Manage. On staging and production, where no workspace runs, Develop becomes a read-only Explore.

Manage your organisation

- **Organisations.** Create an organisation and connect it to its GitHub organisation. The home page shows each organisation with a card per project.
- **Members and roles.** Owners delete projects, clusters and databases, change secrets and settings, manage members and read the audit record. Members work in every project. Owners invite people by email, change roles and remove people, and each action says what will happen. The last owner can never be removed.
- **Signing in.** Single sign-on. An expired session keeps what was typed and offers to sign in again in a new tab.
- **Organisation credentials.** Owners add, rotate and delete the cloud credentials used to create clusters, and connect the organisation's mobile publishing token. Values can be written but never read back.
- **Removing an organisation.** Confirmed by typing its name, refused while projects or clusters remain, with a link to what is in the way.

Manage clusters

- **Environments and clusters at a glance.** The project overview lists every environment and its clusters, each with its health, platform install, applications, services, interfaces and previews.
- **The cluster page.** Readiness, the agent's connection, node capacity in use, activity, domains, and the upgrade channel and maintenance window, all changeable in place. A card shows the agent's command policy and any refused commands.
- **Platform versions.** A page lists every platform component with its version, and warns when a cluster's settings lack something the current version needs.
- **Workspaces.** A page shows who has a workspace on which development cluster. Owners give and remove workspaces, and can let members create their own.

Run a project day to day

- **Changes.** Every open pull request with its author, review state, checks, latest run and preview. Merge is enabled only when the repository's rules allow it, and otherwise says why.
- **Builds and deployments.** Each pipeline run with a step-by-step graph, live logs and its failure report. Logs stay readable and downloadable after the run has finished.
- **Services.** Each service lists its running instances in every environment, with links to its logs, and its resource settings per environment: requests, limits, replicas and autoscaling. Changes are proposed as pull requests.

- **User interfaces.** One page links to every live web interface in each environment, with its running version and previews, QR codes for mobile, and the design review button.
- **Releases.** Promotion, rollout control and rollback, as described in section 2.
- **Configuration.** Each component's settings per environment, with a comparison across environments. Edits become pull requests. On staging and production the pages are read-only and show the exact commit the cluster runs.
- **Data services.** Pages for the database with a data browser, the cache, object storage, the message bus, sign-in settings and users, model serving and mail. Destructive actions need a typed confirmation.
- **Domains.** Each domain's status, setup steps, which environment serves it, the checked switch to another environment, and its history.
- **Kubernetes explorer.** Browse workloads by kind and open their logs.
- **Loops.** Standing automations, each with a goal, guardrails including a spending cap, an evaluator, a run history and an approvals queue. They can start from templates such as an incident responder.
- **Production without a workspace.** Health, releases, configuration, builds, logs and incidents all work on staging and production, where no development workspace exists.

See what is happening

- **What is collected.** Container logs, metrics for every workload and node, request metrics from the mesh, and traces.
- **Dashboards.** Grafana is embedded in the console, with bundled dashboards for API request rates, errors and latency, the database, the message bus and the cache.
- **Health and objectives.** One page shows every cluster in every environment and what its alerts and service-level objectives say. It tells "checked and quiet" apart from "nothing is watching". Objectives are declared in the repository and edited by pull request.
- **Alerts.** Fifteen rules ship by default, covering error rate, slow responses, crash loops, unavailable deployments, memory and CPU pressure, database waits, integration quotas, backup age and restore tests. Each alert shows one state and one next step, and links to its runbook.
- **Incidents, diagnosed by AI.** Each firing alert becomes an incident, and a resolved alert closes it. An AI incident manager diagnoses each one using the project's runbooks and context. The incident page shows the investigation timeline, with actions to acknowledge, resolve, re-run the diagnosis or start a loop.
- **Audit record.** Who deleted, changed, approved or revealed what, across the project and the organisation, filterable by category and person.

Errors

- **Every error, tied to its release.** Services, web apps and mobile apps report errors to a collector on the cluster, tagged with the release and environment. Personal data is removed before an error leaves the app.
- **An errors page per environment.** What is new since the last release, what has come back, what happens most, and how many people are affected.

- **Cause, not just symptom.** Each error links to the release, the pull requests in it and the change most likely responsible.
- **Releases that protect themselves.** A gradual rollout pauses or stops when a release introduces new errors.
- **Fix with AI.** An agent reproduces the error where it can and proposes a fix with a test, as a pull request.

On-call paging

- **Works with the paging service you use.** Grafana IRM, PagerDuty and Rootly, set up in a few steps: connect the account, pick the team or escalation policy from a live list, send a test page.
- **Pages sent from the cluster.** An alert reaches the person on call even when OpenFlame's control plane is unavailable.
- **Kept in step.** Acknowledging or resolving in either place updates the other, and the incident page shows who is on call.
- **Routing rules.** Which alerts page someone and which only notify, by environment and severity, with quiet rules for planned maintenance.
- **A fallback, and honesty.** Email and text messages are used if the paging service is down, and a project with no paging set up is told plainly that nobody will be paged.

Customer-facing status page

- **A public status page per project.** Published to the customer's own Cloudflare account, so it stays up when the project's cluster is down.
- **Real signals.** The components shown are tied to actual health checks and objectives, and uptime figures are computed from real data.
- **Updates a person approves.** Incident updates are drafted in plain language from the incident timeline, and nothing is published without approval.
- **Subscriptions.** People can follow by email, webhook or feed, and planned maintenance can be announced ahead.

Product analytics

- **PostHog, set up for you.** Connected through the project's own PostHog account, with a choice of region. A self-hosted PostHog can be used by changing one setting.
- **Instrumented from the start.** Generated web, mobile and backend code reports a named, documented catalogue of events kept in the repository.
- **Private by default.** Consent and opt-out are respected, session recording is off unless switched on and never captures screens holding personal data, and analytics are off for minors' accounts.
- **Funnels, retention and experiments** are done in PostHog, linked from the console and connected to feature flags.

Integrations with outside services

- **One gateway.** Calls to outside services go through a gateway on the cluster, which applies each integration's rate limit, quota and budget and shows live usage.
- **Does each integration work?** A real, read-only test for thirteen kinds of integration, reporting one honest state for each.

Control cost

- **Capacity in view.** Each cluster page shows what its nodes can provide, what is requested and what is in use. Section 6 covers cost in full.
- **Budgets on automation.** Every loop has a spending limit, every integration a budget and quota, and AI incident diagnosis is metered and stops at its cap.
- **AI subscriptions.** A page lists the AI accounts that workspaces run on. Building uses the team's own subscriptions, not a metered platform key.

5 Secure

Security is built into each stage: what goes into an image, what may run on a cluster, what may talk to what, and who may see or change anything.

Supply chain

- **Signed images with provenance.** Builds have no right to push. A separate step pushes each image, signs it with the project's key, attaches a record of how it was built and a bill of materials, and records it in a ledger.
- **Only signed software runs.** Each cluster checks every new workload's images and charts against its trust list, first in report mode and then enforcing. Refusals appear in the console with one fix each, and an owner can allow a single image for up to 90 days.
- **Project and platform keys.** Created by the control plane, private parts kept only in the secret store, and rotated from the console in steps.
- **Builds with the network off.** Dependencies are fetched first, then code compiles without network access. A proxy can limit builds to approved registries.
- **Build code kept from build secrets.** A project's own code runs in the pipeline under an account with no cluster access, and policies stop pipeline steps from reading secrets they should not.

Vulnerability and dependency monitoring: Firewatch

- **Continuous scanning.** Dependencies, container images, running workloads, infrastructure code, leaked secrets, licences and end-of-life software. Findings are ranked by whether they are being exploited, how likely that is, severity and whether the code is reachable.
- **One fix path per finding.** Each finding has one status, one explanation and one action. Components are upgraded to the lowest version that fixes the problem.
- **Update pull requests, never merged automatically.** Updates arrive as pull requests linked to the findings they fix, and a person merges every one.
- **Rules against bad packages.** A waiting period before newly released versions are adopted, signals for suspicious packages, and a search across repositories, images and running workloads.
- **Runbooks for a compromised key or package.** Declare the incident, see what is affected, and rotate or replace step by step.

Inside the cluster

- **Admin tools behind sign-in.** Dashboards, the database console and other operator tools accept only members of the project's organisation, and refuse when the check cannot be made.
- **Admin ports locked to named callers.** The database's admin interface and the sign-in system's admin interface accept only the specific workloads that need them.
- **Deny by default.** Network traffic between parts of the cluster is refused unless a rule allows it.

- **Encrypted, identified traffic.** The mesh gives every workload an identity and encrypts traffic between them.
- **Narrow message-bus rights.** Each service connects to the bus as its own user with only the rights it needs, and anonymous connections are refused.
- **Isolated workspaces.** A developer's workspace admits only its owner.

Secrets and access

- **Secrets per environment.** Kept in the secret store, with create, reveal, delete and a guided rotation. Generated keys are created without ever being displayed.
- **Short-lived access.** OpenFlame reaches a cluster with short-lived credentials from the cloud, never a stored administrator file.
- **Everything recorded.** Sensitive actions appear in the audit record, and every command sent to a cluster is signed and logged on the cluster itself.

Compliance evidence

- **Controls mapped to evidence.** A catalogue of controls for SOC 2, ISO 27001 and HIPAA, each tied to what proves it: signed builds, the audit record, access controls, policies in force, backups, restore drills and change history.
- **Collected continuously.** Each control shows as evidenced on a date by a named source, failing, or the customer's own to provide, such as written policies and training, which can be attached.
- **An evidence pack for the auditor.** Exported for a chosen period, and tamper-evident: a daily fingerprint of the evidence record is published to a public transparency log.
- **Alerts when a control lapses.** For example a restore drill that is overdue, or a policy switched from enforcing to reporting.
- **Access reviews.** A new owner is acknowledged by a second owner, and access is reviewed every quarter.

GDPR

- **Evidence by article.** GDPR is a law, not a certificate, so OpenFlame shows what can be evidenced toward each obligation, by article number, and what remains the customer's to provide.
- **A record of processing.** Generated from the data model's personal-data labels, the integrations in use and where data lives, and kept current as the system changes.
- **People's rights.** Export and erasure of one person's data across every service, with a log showing each request and whether it was met in time.
- **Breach handling.** Each incident records when it was detected and shows the 72-hour notification clock, with a place to record the notification made.
- **Privacy by design and default.** Personal data must be labelled before a data model is merged, and analytics and session recording are off by default for minors.

- **Processors.** A generated list of every outside service that receives personal data, with the customer's agreements attached, and a plain statement of what OpenFlame's own control plane processes.
- **Transfers and retention.** Transfers outside the declared region are shown from the residency page, and the retention declared for each kind of data is checked against what the system actually deletes.
- **Consent records.** Where an application relies on consent, what was agreed and when is kept.
- **An impact assessment template.** Pre-filled with the system's facts for the customer to complete.
- **Yours to provide.** The lawful basis for each use of data, the privacy notice, and a data protection officer where required, each with a place to attach it.

Data residency

- **Declare where data must stay.** A project states its region, such as the EU.
- **Everything checked against it.** Cluster regions, database and backup locations, file storage, the disaster recovery standby, logs, outside integrations and AI providers. A location that cannot be established counts as failing.
- **Refused at the start.** A cluster, backup destination or standby in the wrong region is refused when it is created, with the reason.
- **A residency page.** Every place data lives, with the evidence for each, including exactly what the control plane holds and where it is.
- **Alerts on drift.**

6 Cost management

OpenFlame knows the exact shape of every cluster it runs, hour by hour, and can create a cluster on any of seven clouds. That lets it say what a workload costs today and what the same workload would cost somewhere else.

What you are spending

- **A cost for every cluster.** A monthly figure built from the cluster's real shape over time: machine sizes, node counts as they scale, disks, storage, load balancers, managed databases and region.
- **Estimate or actual, always labelled.** A figure is either an estimate from the cloud's published prices or taken from the customer's real bill once billing data is connected. Each one says which, the date of the prices used, and what it leaves out.
- **No guessed numbers.** A price that cannot be obtained is shown as missing, never filled in.
- **Cost per service.** Each service is charged for the larger of what it requests and what it uses. Platform overhead and idle capacity have their own lines, so they are not hidden inside services.
- **Views at every level.** A cost page per project, a cost line on each cluster, a roll-up across the organisation, and cost per environment.
- **Bill access kept separate.** Reading the customer's bill uses its own optional, read-only permission, apart from the permission used to create clusters.
- **Resources tagged for attribution.** Every cloud resource a cluster creates carries the same cost tags, so the real bill can be traced back to cluster and environment.

The same workload on another cloud

- **A like-for-like comparison.** The cluster's actual shape over the last 30 days is priced on each of the other clouds, with each machine size mapped to its closest equivalent in a comparable region.
- **The working shown.** One column per cloud with the total and the main lines. The size mapping and every assumption are visible, so the comparison can be checked.
- **Honest about what cannot be compared.** Data transfer charges, managed-service differences and minimum sizes vary between clouds, and the comparison lists them.
- **Pricing options.** On-demand prices by default, with spot pricing and one- and three-year commitments as options, and currency conversion at a dated reference rate.
- **Right-sizing.** A second comparison shows what the same workload would cost on the current cloud if the nodes matched what is actually used.
- **From comparison to a move.** The comparison links to the steps for moving an environment to another cloud: create the cluster, restore the data, switch the domain.

Budgets and other spending

- **Budgets with alerts.** A budget per project or environment, with a forecast and a notification when spending is heading over.

- **AI spend.** Token use and estimated cost for agent runs, reviews, incident diagnosis and loops.
- **Pipeline and preview cost.** What builds cost per project, what each preview environment costs, and which previews and workspaces are sitting idle.

7 Business continuity

Because the same platform installs on every cloud and an application deploys from its repository, a project can be kept ready to run on a second cloud, and brought up there when the first one fails.

A continuity plan for each environment

- **Declared targets.** Each environment states how much data loss and how long a recovery it can accept. The plan is a file in the project's repository.
- **A live status page.** It says truthfully whether the targets are being met right now, from backup age, sync lag and the last drill: protected, behind target, never drilled, or standby not reachable.
- **Only measured numbers.** A recovery time or data-loss figure is shown only when a drill measured it. Otherwise it is labelled a target.
- **A runbook that is always current.** Generated from the system as it is and kept in the repository.

Levels of protection

LEVEL	WHAT IT MEANS
Backup to a second cloud	Backups and files are kept on another provider. A cluster is created there and restored only when needed. Lowest cost.
Pilot light	A small cluster already runs on the second cloud with the platform installed and the database kept nearly current. Recovery is much faster.
Warm standby	The whole application runs on the second cloud, scaled down, ready to take over.

- **Priced before you choose.** Each level shows what it costs to run, using the same prices as the cost comparison.

Keeping the second cloud in sync

- **Database.** Continuous backups go to storage on the second provider, and a standby database can follow them to stay nearly current.
- **Files.** Uploaded files are copied one way to a bucket on the other provider. A deletion is never copied across immediately, which protects against mistakes and attacks as well as outages. After a restore, files and database are reconciled to the same moment.
- **Secrets.** An encrypted copy of the environment's secrets is held under a key only the customer has, so recovery does not depend on OpenFlame being available.
- **Releases and software.** The standby follows the same release from the repository, and the images and charts it needs can be copied to a registry the customer owns.

- **The same accounts.** A standby is a second cluster of the same environment, so people's accounts and sign-ins carry over.

Failover and return

- **One guided action.** Failover cuts the failed cluster off so it can never accept writes again, promotes the standby database, brings the application up, and moves the domain. It works even when the failed cluster cannot be reached.
- **The address does not change.** The custom domain moves to the standby, so web and mobile users keep the same address.
- **Works without OpenFlame.** A command-line path performs a recovery using only the customer's repository, cloud accounts and escrowed secrets.
- **A planned way back.** Failback syncs in the other direction and switches back in a chosen window.
- **Typed confirmation.** Failover and failback name the environment and cluster and must be confirmed by typing.

Drills and readiness

- **Scheduled recovery drills.** A drill really restores onto the other cloud, runs the project's own checks against the result, measures the recovery time and data loss, records them, and removes what it created. Its cost is shown beforehand.
- **Alerts.** A failed drill, or targets that are no longer being met, raise an alert.
- **Portability checks.** A standing check reports anything in a project that has become tied to one cloud, before it gets in the way of a recovery.
- **The control plane is covered too.** OpenFlame's own control plane keeps off-site backups that cannot be deleted and can be rebuilt, so it is not a single point of failure for anyone's recovery.

8 The service store

Most applications need the same building blocks: background jobs, notifications, messaging, payments, search. The store provides them as finished services an application installs, so they do not have to be generated or written again.

How it works

- **Install or fork.** An application can run the published version of a service, or take its source code and change it. The code is open source under the Apache 2.0 licence, written in Go.
- **One copy per application and environment.** Each installed service gets its own tables and its own database role inside the application's database. Nothing is shared between applications.
- **The planner knows the catalogue.** When a new system is planned, store services are chosen where they fit.
- **Set up by the platform.** Installing a service provisions what it declares it needs: database access, message-bus rights, signing keys, storage, routes with access rules, and credentials for the other services it calls.
- **Provider keys stay out of services.** Keys for outside providers of push, mail, text messages, AI models and payments are held by the project's runtime, which also decides what may leave the cluster. A store service never holds one.
- **Upgrades arrive as pull requests.** A newer compatible version is proposed as a pull request that changes only the version, quotes the changelog and states the migration checks. Nothing merges by itself, and a new major version is offered, not proposed.
- **A catalogue per platform version.** A deployment refuses a version of a service that the cluster's platform does not carry.
- **Permissions checked on the server.** Every service checks who is calling on every request, and refuses when it cannot tell.

Core services

SERVICE	WHAT IT DOES
Jobs	Background work: queues, retries, schedules, and heavy jobs handed to the workflow engine. A job runs at least once and a schedule fires once per due time.
Identity	A person record in front of the sign-in system, with stricter defaults for minors. Also teams: groups, roles, invitations, and "may this person do this?" checks for other services.
Notifications	One front door for telling a person something. It picks the channel by their preferences and falls back in order: in-app, then push, then mail or text message. Handles opt-outs and unsubscribe links.

Realtime	Live connections to browsers and phones: topics, presence, and who may subscribe. Browsers never touch the message bus directly.
Conversations	Messaging with one thread per person across every channel: chat, text messages, email, calls and voice agents. Direct, group and business inboxes, with replies, reactions, read state and staff-only notes. End-to-end encryption is available for chat.
Media	Uploads straight from the browser, image resizing, video handling, and signed links so files are only reachable by people allowed to see them.
AI interface	One way for an application to call a model. The runtime holds the keys and applies limits, so the application never has a provider key.
Search	Search over the application's records, on its own database. Permissions are part of the query, so nobody sees results or counts they are not allowed to.
Payments	Subscriptions, one-off purchases, entitlements, invoices, refunds and disputes, for a person or a group. Never sees a card number.
Activity record and privacy tools	A record of who did what, plus exporting and erasing one person's data across every service.
Moderation	Reports, review queues, blocking and automated checks on user content.
Documents	Generating documents from templates, such as invoices and contracts, and collecting signatures.
Developer access	API keys, usage limits and webhooks, so an application's own customers can build against it.
Location	Places, distance and "what's near me" queries, with live position sharing through Realtime.

Services for specific kinds of application

SERVICE	WHAT IT DOES
Commerce	Catalogue, cart, orders and stock.
Auction	Listings, bidding, time extensions and settlement.
Bookings	Availability, appointments, reminders and cancellations.
Reviews	Ratings and reviews tied to a real purchase or booking.

Cluster add-ons

These are switched on for a cluster, not installed per application.

ADD-ON	WHAT IT DOES
File transfer server	SFTP for partners who need it.
Media server	Carries the audio and video for calls and voice agents.

9 Built for agents

Everything OpenFlame does can be driven by an AI agent as well as by a person. Through the Model Context Protocol, an agent in any compatible tool can create a project, build it, ship it and look after it, within limits a person sets.

Two servers, one hand-off

SERVER	WHAT AN AGENT CAN DO WITH IT
OpenFlame MCP	Runs on the control plane. Create and launch projects, request clusters, and read the state of everything: environments, releases, health, cost, compliance and continuity.
Project MCP	Runs on a project's development cluster. Plan, generate, review, run tests, debug, read logs and errors, and propose changes, inside that one project.

- **A clean hand-off.** Creating a project returns its project server's address and a token that works only there, so an agent moves from "make me a project" to working inside it without a person in between.
- **Any agent, any tool.** The servers speak an open protocol, so a developer can use the coding agent they already prefer.
- **Tools shaped around tasks.** A small set of meaningful actions, such as launch a project, propose a change, promote a release or diagnose an incident, rather than hundreds of low-level calls, with read access to everything.
- **One implementation.** The servers use the same interfaces as the console, so an agent and a person always see and do the same things.

Limits a person sets

- **An agent's authority is narrower than its owner's.** Each agent works with a token scoped to one project or organisation, which can be revoked at any time.
- **Spending needs approval.** Creating a cluster, or anything else that costs money, asks a person first by default, and every agent has a budget and a cap.
- **Destructive actions stay with people.** Deleting a cluster, a database or a project is never available to an agent.
- **Production by pull request.** On staging and production an agent can read and can propose a change as a pull request, which goes through the same reviews and approvals as any other. It never acts there directly.
- **Everything on the record.** Each action is recorded as a named agent acting on behalf of a named person, and every command sent to a cluster is signed and checked against the policy in the customer's repository.

- **Guarded against manipulation.** Because an agent can be misled by what it reads, scoped tokens, approvals and budgets apply however the request arrived.

10 Ownership and no lock-in

OpenFlame is built so that a customer can leave with everything, and so that the application keeps running if they do.

- **Your cloud account.** Clusters are created in the customer's own account, on the customer's bill, under the customer's control.
- **Your repository.** Code, deployment files, pipeline definitions, environment settings and the record of every release live in the customer's repository.
- **Your data.** The database, files, cache and sign-in system run on the customer's cluster. Backups go wherever the customer chooses.
- **Your domains.** Custom domains live in the customer's own Cloudflare account.
- **Open-source components.** The stack is assembled from widely used open-source software, named in section 3, and the store's services are open source.
- **The application runs without OpenFlame.** Removing the cluster agent leaves the application running exactly as it was.
- **A written list for leaving.** Every dependency on OpenFlame's control plane, such as the secret store, the registry, the charts, the built-in domain and the tunnel, is listed with what replaces it. The list is kept current as features are added.
- **Nothing reaches in.** The control plane never connects into a customer's cluster. The cluster collects signed commands and checks each one against a policy the customer keeps in their own repository.